

Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Tomáš Jedlička

Utilizing HLA for agent based development platforms

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Tomáš Plch

Study programme: Informatics

Specialization: Software Systems

Prague 2011

I would like to thank to my supervisor Mgr. Tomáš Plch for his help with writing of this thesis.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date 8.12.2011

Topic

| | |
|--|----|
| 1. Introduction | 11 |
| 2. High Level Architecture | 13 |
| 2.1 Communication between federates | 14 |
| 2.2 Synchronization of federation | 16 |
| 3. Analysis | 19 |
| 3.1 Synchronization | 20 |
| 4. Analysis of HLAProxy architecture | 23 |
| 4.1.1 Common Interface | 24 |
| 4.1.2 The RTI Communication Module | 24 |
| 4.1.3 The RTI Support Module | 25 |
| 4.2 FOM Abstraction Module | 27 |
| 4.3 Typical use cases | 27 |
| 4.4 Selection of the RTI | 28 |
| 5. HLAProxy internals | 31 |
| 5.1 Common Interface | 31 |
| 5.1.1 Plug-in internal structure | 32 |
| 5.1.2 Component registration and instantiation | 33 |
| 5.2 RTI Communication Module | 34 |
| 5.2.1 HLA Federate and its configuration | 34 |
| 5.2.2 Object class support | 35 |
| 5.2.3 Interactions | 36 |
| 5.3 RTI Support Module | 37 |
| 5.3.1 Database interface | 39 |
| 5.3.2 Request scheduler | 39 |
| 5.3.3 Data storage | 46 |
| 5.3.4 Worker threads | 50 |
| 5.4 FOM Abstraction Module | 52 |
| 5.5 Data shared library | 54 |
| 5.5.1 HLA object instances | 55 |
| 5.5.2 The HLA read-only object instances | 56 |
| 5.5.3 The HLA interaction classes | 56 |
| 5.5.4 Value marshalling and unmarshalling | 57 |
| 5.5.5 Generating source code | 58 |
| 5.5.6 HLA private interfaces | 59 |
| 5.5.7 The Database private interfaces | 59 |

| | |
|--|----|
| 5.6 Application specific support module implementation | 60 |
| 5.6.1 Component for configuration management | 60 |
| 5.6.2 Output logging component | 62 |
| 6. Available engines | 65 |
| 6.1 Virtual Battle Space 2 (Real Virtuality engine) | 65 |
| 6.1.1 LVC Game | 65 |
| 6.2 Source Engine | 66 |
| 6.3 Unreal Engine | 67 |
| 6.4 Cry Engine 3 | 68 |
| 7. Integration to the Source Engine | 69 |
| 7.1 Integration of HLA Proxy middleware | 69 |
| 7.2 Synchronization | 70 |
| 7.3 Output logging | 71 |
| 7.4 Remote controlled NPC | 72 |
| 8. Integration to the CryEngine 3 | 73 |
| 8.1 Integration of HLA Proxy middleware | 73 |
| 8.2 The Player entity | 74 |
| 9. Real deployment scenario | 75 |
| 9.1 HL2 federate | 75 |
| 9.2 CryEngine 3 federate | 76 |
| 9.3 HLA-client federate | 76 |
| 10. Performance | 79 |
| 10.1 Measurements | 80 |
| 10.1.1 Write throughput of the database | 81 |
| 10.1.2 Multiple reads without writes | 81 |
| 10.1.3 Mixed load | 82 |
| 10.1.4 Randomized operation throughput | 82 |
| 10.1.5 Huge amount of attributes | 83 |
| 10.1.6 Conclusion | 83 |
| 11. Related work | 85 |
| 12. Discussion | 87 |
| 12.1 Future work | 88 |
| 13. Bibliography | 89 |
| Appendix A. List of examples | 91 |
| Appendix B. List of tables | 93 |
| Appendix C. List of figures | 95 |
| Appendix D. Introduction to C++, ABI and Runtime libraries | 97 |

| | |
|--|-----|
| D.1. Linking in the middleware | 97 |
| D.2. Dynamically or statically linked runtime library | 98 |
| D.3. Calling conventions | 99 |
| D.4. How to make API calls safe in mixed runtime environment | 100 |
| D.5. Binary compatibility | 103 |
| Appendix E. Setup MAK RTI | 105 |
| E.1. Microsoft Windows | 105 |
| E.2. Fedora 16 | 105 |
| Appendix F. Building of the middleware | 107 |
| F.1. Windows build environment | 107 |
| F.1.1. Building boost libraries | 107 |
| F.1.2. Altova XSLT processor | 107 |
| F.2. Building the middleware on Windows | 107 |
| F.3. Linux build environment | 108 |
| F.3.1. Saxon XSLT processor | 108 |
| F.4. Building the middleware on Linux | 108 |
| Appendix G. The HLAProxyMod | 111 |
| G.1. Building of the mod | 111 |
| G.2. Running the mod | 111 |

Název práce: *Utilizing HLA for agent based development platforms*

Autor: Bc. Tomáš Jedlička

Katedra / ústav: Kabinet software a výuky informatiky, Univerzita Karlova v Praze

Vedoucí diplomové práce: Mgr. Tomáš Plch

e-mail vedoucího: tomas.plch@gmail.com

Abstrakt: High Level Architecture (HLA) poskytuje univerzální řešení pro propojení více simulačních prostředí a aplikací, vytvářející složitější simulační system. Základní myšlenka HLA je kontrolovaná a směrovaná výměna objektů a událostí (prostřednictvím RunTime Interface - RTI) sdílených jednotlivými účastníky systému (např. simulacemi). Vzniká tak distribuované datové prostředí. Cílem práce je zkoumat využitelnost HLA pro vývojové platformy pro virtuální agenty (např. Pogamut). Dalším cílem práce je poskytnout jednoduchý a transparentní přístup k HLA pro aplikace, které neimplementují HLA. Součástí práce je popis architektury a popis prototypové testovací implementace, která integruje HLA do dvou různých uzavřených herních enigmů poskytujíc jim tak schopnost komunikovat s jednoduchou klientskou aplikací užívající rozumnou podmnožinu HLA standard. Součástí práce jsou měření výkonu naší implementace

Klíčová slova: High Level Architecture, Herní Engine, Runt Time Interface, IEEE 1516-2010, Distribuovaná databáze

Title: *Utilizing HLA for agent based development platforms*

Author: Bc. Tomáš Jedlička

Department / Institute: Department of Software and Computer Science Education,
Charles University in Prague

Supervisor: Mgr. Tomáš Plch

Supervisor's e-mail address: tomas.plch@gmail.com

Abstract: The High Level Architecture (HLA) provides a universal solution for inter-connecting various simulation environments and applications thus creating a more complex simulation entity. The idea is built upon controlled and directed data exchanges of objects and events (via the RunTime Interface – RTI) shared by participants (i.e. simulations) thus creating a distributed data environment. The aim of this thesis is to investigate usability of HLA for agent based development platforms (e.g. Pogamut) as well as providing transparent and simple to use access to HLA for HLA unaware applications. The thesis describes architecture and provides a prototype proof-of-concept implementation, which integrates HLA with two different (closed source) game engines providing them the ability to communicate to a simple client application according to a reasonable subset of the HLA standard. The thesis also provides performance measurements of prototype implementation.

Keywords: High Level Architecture, Game Engine, Run Time Interface, IEEE 1516-2010, Distributed Database

1. Introduction

Computer simulations play an important role in today's world. They are used for training purposes, because they are cheap (you can learn to fly simulated plain instead of real one) and can be deployed on a larger scale. Today computer games are small simulations as well, creating a synthetic world of their own.

Unfortunately simulations have significant problems. As demand for simulation increases it is not possible to find a single simulation capable of handling all requirements due to limited hardware performance. It may be impossible and impractical to design and implement a complex simulation in reasonable time. It may be too expensive.

The High Level Architecture (HLA) has been designed to overcome such problems by defining an abstract communicational protocol and universal simulation software architecture. HLA design allows composition of small and simple simulations to more complex ones. Smaller simulations can cooperate in modeling large synthetic worlds. The HLA architecture design allows reusing older HLA aware existing simulations.

The following Figure 1-1 presents the basic concept of HLA – creating a communicational platform for different simulations. However HLA does not define interconnection between simulations only. It is also capable to join human controlled entities (Live participants) into synthetic world. It is also possible to use monitoring tools that do not actively in the simulation at all and are only observers of the simulated environment (Passive viewers). The actual interconnection is created and maintained by the Run Time Infrastructure (RTI).

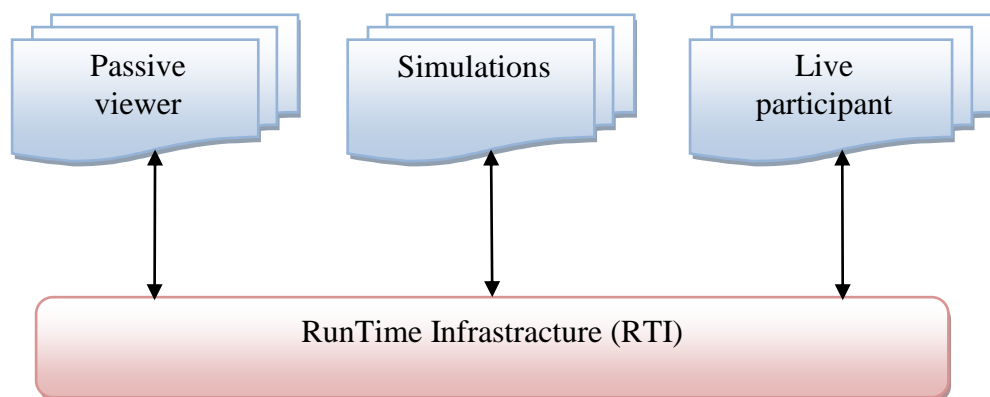


Figure 1-1 HLA interconnecting various environments over RTI

An agent based development platform can be perceived as a virtual environment (i.e. simulation) which is inhabited by artificial beings capable to interact with the synthetic world and each other. Good examples of agent based development platform are today's computer games. The prototyping tool is software, which provides a development and deployment environment for decision making mechanism for agents and their artificial intelligence. A key feature is the provided virtual environment where either the tool provides its own simulation or connects to a development platform (i.e. computer game). Presently, the usual scenario is that a prototyping tool is connected to agent based development platform and takes over control of some

artificial beings [1]. The prototyping tool than simulates an artificial being brain and performs interactions with synthetic world (simulated by connected agent based development platform).

This thesis aims on investigation of usage and prototype implementation of HLA middleware that can transparently enrich 3rd party application with HLA capabilities. The aim is to allow easy enhancements of existing agent based development platform with support for HLA. This will allow usage of standardized communication between agent based development platform and prototyping tool. HLA enabled prototyping tool is easily reusable with other HLA enabled agent based development platforms.

Main motivation for writing this thesis came from the KSVI department of the MFF UK. The department is actively developing project called *Pogamut*[2]. The *Pogamut* project provides development environment for artificial intelligence and its testing and debugging in simulated environment based on the Unreal technology (Unreal Tournament 2004).

As the *Pogamut* evolves, one simulation environment is not enough. The *Pogamut* development team is actively working on developing new connections to other platforms (e.g. Unreal Development Kit[3], Defcon[4]). Usage of the HLA can provide a general purpose solution for interconnecting any arbitrary applications. This thesis has following goals::

1. Investigate usage of HLA in agent based development platforms.
2. Provide simple to use solution that enables HLA functionality in 3rd party applications.
3. Implement a proof-of-concept prototype.
4. Perform real-case study on today's available agent based development platforms

The thesis structure is as follows:

- The first chapter describes brief introduction.
- The second chapter contains brief description of HLA.
- The third chapter contains problem statement and analysis.
- The fourth chapter contains analysis of architecture of the prototype.
- The fifth chapter contains discussion about prototype implementation
- The sixth chapter contains discussion about various available agent based development platforms.
- The seventh chapter contains discussion about integration into the Source Engine [5].
- The eighth chapter contains discussion about integration into the CryENGINE 3[6].
- The ninth chapter contains description about real case study.
- The tenth chapter contains performance measurements and discussion.
- The eleventh chapter contains discussion of related works.
- The twelfth chapter contains final conclusion.

2. High Level Architecture

High Level Architecture (HLA) was designed to solve the U.S. Department of Defense's (DoD) functional needs for environment modeling and simulation. The idea is based on simple assumption that no simulation can solve all the DoD's functional needs for simulation integration. The technical complexity of creating a single application which can satisfies all requirements is beyond of what is possible to be done today. It is required to design complex simulations by composing of smaller ones. The "small" simulation has to be designed to allow easy composition and reuse. It is also needed to overcome the problems caused by changing state of technology. HLA defines abstract communicational language and application architecture that allows data exchanging in a way that overcomes all mentioned complications.

In fact HLA is not used only for composition of multiple simulations together. The army uses for example real sniper rifle models with software that is HLA compliant. This allows combining simulation software with real entities, which are controlled by human actor (live participant), to create a rich and complex training environment. It is possible to connect applications that do not participate in complex simulation at all. They are just passively monitoring the state of the whole simulation (passive participants).

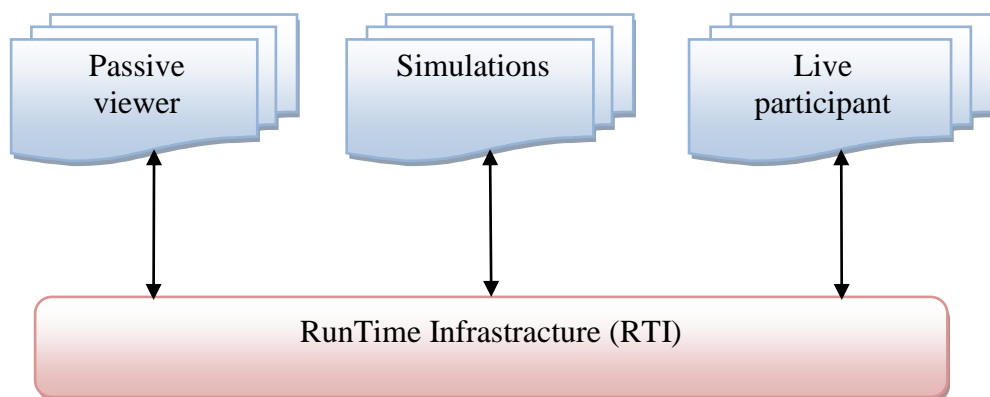


Figure 2-1 Basic concept of HLA

The Figure 2-1 shows the basic use case of HLA. The HLA introduces *Run-Time Infrastructure* (RTI) communicational middleware. All participants (in HLA terminology *federates*) are required to talk with each other only through the RTI. It doesn't matter whether the federate is a simulation, live participant or just passive viewers observing the world and its entities.

The complex simulation composed from all participating federates is called the *federation*. The HLA defines also specific term the *federation execution*. The term federation execution means some federation composed of specific federates that is actively performing some simulation over time.

The HLA definition is composed from three standards:

- HLA Rules (IEEE 1516[7])
- HLA Federate Interface Specification (IEEE 1516.1[8])
- HLA Object Model Template (IEEE 1516.2[9])

2.1 Communication between federates

Communication between federates has a specific complex protocol. Federates are not allowed to communicate with each other directly. Only allowed communication is through the RTI by invoking its services. The RTI can also communicate with federate by invoking federate's services. The full list of available services and their usage is described in IEEE 1516.1-2010[8] standard. This is the only way of how federates can interact with each other.

Federates can exchange data with each other. . The HLA defines two types of data that are transferred between federates:

- Object class instances
- Interactions

An object class instance represents entity in a distributed simulation world. Every instance must have its name that is unique among all federates participating in federation execution. That means that all federates see that instance only once. Every object instance is instance of specific object class. The class structure is depicted on Figure 2-2. Every object class can define a set of attributes that will be present in

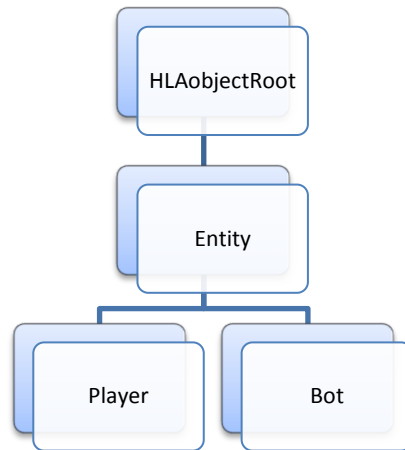


Figure 2-2 Object class hierarchy

every instance of that class. The new object class can be derived from one of existing classes. For example the *Player* class is derived from the *Entity* class. In that case every instance of *Player* contains all attributes defined by *Player* class and all attributes defined in its super class (the *Entity*). The root class is always *HLAobjectRoot* that has one attribute *HLAPrivilegeToDeleteObject*. As a result all classes inherit this specific attribute. The meaning of the attribute will be explained later.

The communication between federates is based on publisher/subscriber model. At any time at most only one federate can publish new values for an attribute, but there can be multiple federates subscribed for receiving values of that attribute. The federate that can publish new value of an attribute is being called attribute owner. The HLA defines various methods of how the owner of the attribute can be changed. An object class attributes might be owned by multiple federates. This is called cooperative modeling of an entity.

The HLA also defines mechanism for creation, deletion and discovery of new instances. If federate creates new object class instance then all other subscribers of that class attributes are notified about new instance. The deletion of an object class

instance can be done only by federate that owns the *HLAPrivilegeToDeleteObject* attribute.

The HLA discover mechanism is designed to allow communication between new and older federates. The Figure 2-3 shows the idea of development of backward compatible federates in respect of the data model. Federate A has old data model (represented by left tree) but federate B has new extended data model (represented by right tree). Federate A subscribed to receive all updates of class *Player*. The federate B creates only instances of *Alien* or *Human*, which contain additional details in its additional attributes. The federate A is not capable to work with instance of class *Alien* or *Human*, because it does not know anything about them. The HLA is designed to solve this issue and thanks to the inheritance it will automatically cast all *Alien* and *Human* instances to type *Player* for federate A. This design allows reusing of older federates as long as they know at least subset of hierarchy of object classes.

The last thing that remains to be explained is definition of attribute values. The HLA does not define any support for attribute values. The RTI transfers values as a set of bytes and does not care about their contents. It is up to federate developers to use common well known encoding. The IEEE 1516.2-2010[3] defines standard encoding for some types, but it can be extended to suit the specific federation needs.

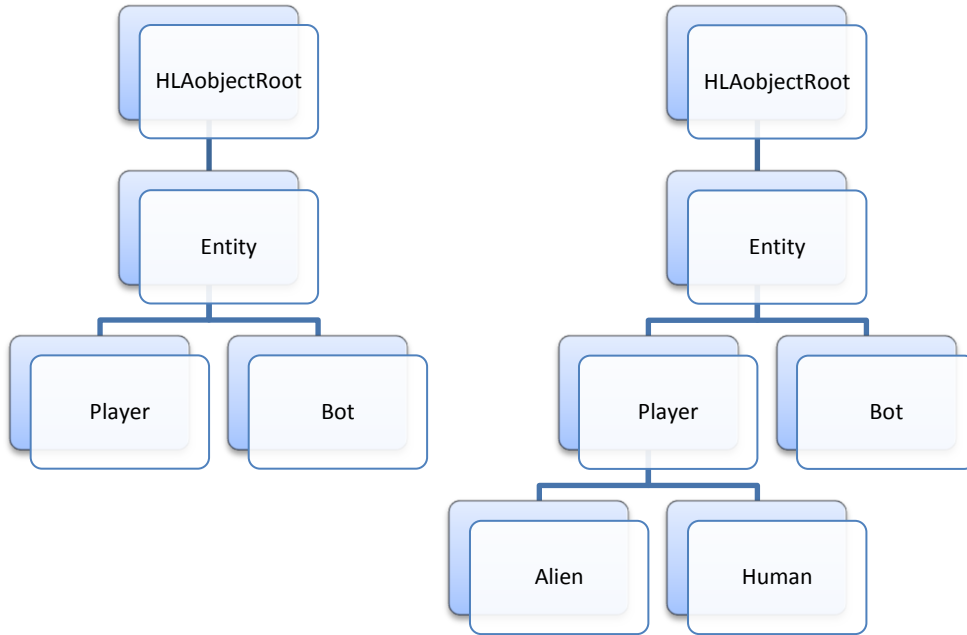


Figure 2-3 Backwards compatible object model

The second type of data transferred during federates communication are *interactions*. Interactions share same idea as design object classes. Every interaction is instance of some interaction class that describes its parameters. The interaction classes can be derived from each other in the same way as objects. The only one difference is that the root class has name *HLAinteractionRoot* and does not have any members. Interactions are backwards compatible with older federates too. The backward compatibility is realized in the same way as for object instances. When an interaction instance is received, it is automatically casted to known subscribed interaction class by the RTI for older federate (in exactly the same way as object classes are casted to one of its parent classes). Differences between interaction and object class instance are by its semantics. An interaction class instance represents an event that has occurred in the simulation world. Interactions are sent with filled in parameter values and its parameters cannot be updated or written to. The interaction delivery

works again on publisher/subscriber model basis. One federate publishes interaction and all subscribers will receive it.

Every federation's data model has to be specified in the document which is called Federation Object Model (FOM). The FOM contains all information about all classes and additional details (e.g. attributes, parameters, transportation type ...). The FOM contents are defined in [9]. Another type of document is Simulation Object Model (SOM). The SOM document has identical contents as FOM document, but it contains data object model of specific simulation. The SOM document is used as a description of simulation capabilities that can be used by federation designers when designing new federations. The SOM details are left out, because it is out of scope of this thesis.

The FOM is important for the RTI as it allows or disallows usage of some services or can be used for network traffic optimizations (network transfer optimizations are defined in Data Declaration Management section in IEEE 1516.1-2010[8] standard and are out of scope of this thesis). The FOM contains information about desired transportation type for attribute values (best-effort or reliable). For example the object class can be marked as subscribe only or publish only. The RTI needs to know this, because it can optimize network traffic and also deny usage of some services on incompatible attributes. The detailed description can be found in the Object Model Template (OMT) section in HLA standard.

2.2 Synchronization of federation

The next important task in HLA design is to allow time synchronization in a federation. The HLA defines universal solution for time synchronization between federates and every federate can decide how it will participate in time management. The available options are:

- Time regulating management
- Time constrained management

Every federate in federation can choose to use both, one or none from the above. The basic idea is that HLA has the internal notion of logical time and the simulation's advance depends on the time management choice made. A time constrained federate must ask through the RTI before being allowed to advance its logical time to a new value. It can't advance unless the RTI approves. A time regulating federate confirms time advancement requests.

The FOM can define the ordering type for an object class attribute or an interaction class parameter. Values updates are then transferred in receive order (that means no order) and time stamped order (strictly ordered according to its time stamp). The selection of time participation of a federate has impact on delivery order of attribute updates and interactions. Federates that do not participate in time management are not able to receive attribute updates or interaction sorted according to their timestamps even that the FOM states that the most desired transfer type is strictly ordered by timestamps.

This thesis uses federates that are using both time regulating management and time constrained management. Combination of both options is best for real-time synchronization, because both federates must wait for each other before advancing their logical time. Therefore they are always synchronized.

The type that represents logical time must be same across a federation (for example float). However federates can choose whether they see time as a continuous function or discrete time steps.

The description of time management presented here is a very brief overview of the HLA time management. The HLA defines exact steps that have to be performed for advancing of time to guarantee the correctness of delivery of attributes updates and interactions. The full detailed description can be found in IEEE 1516.1-2010[8]

The synchronization of time itself is not enough to synchronize all federates. From time to time it is required to be sure that all federates are in some state. For this purpose the HLA defines *synchronization points*. A federate that is initiating synchronization announces new synchronization point and all other federates are noticed about the new synchronization point. Every federate that is at the time of announcement participating in federation execution must announce that it reached the synchronization point. Then the RTI notifies all federates that a federation is synchronized and federates continue in normal work again.

This type of synchronization is for example used during federation startup when it is required that all federates join the federation and start working together at the same time.

3. Analysis

An agent based development platform (e.g. Unreal Tournament 2004[3]) can be perceived as a kind of simulation where artificial beings interact with a synthetic world. The prototyping tool (e.g. Pogamut[2]) is software that allows development and deployment of decision making algorithms in a connected development platform. The usual scenario is that a prototyping tool is connected to agent based platform and takes over control of some artificial beings. An agent based development platform must provide information about synthetic world (e.g. entities, events ...). A prototyping tool can interact with an agent based platform by performing modifications to the synthetic world state (e.g. moving of an artificial being in the world).

The main problem of Pogamut prototyping tool is limited abstraction of its communication with agent based development platform. Every agent based development platform is a bit different and therefore it is required to design and implement new connection mechanism that suits the agent based development platform. This thesis aims on investigation of usage of HLA in today's game engines which can be used as agent based development platforms. The usage of HLA could provide universal solution for interconnecting agent based platforms with prototyping tools. HLA data model can introduce new level of abstraction and developers of prototyping tool can stay focused on the tool itself and not spend long time working on communication with agent based development platform.

However the problem which we address in this thesis is more general. The aim is to design a universal solution that can be used for enabling HLA support in any 3rd party application. The application does not need to be designed for any network support at all.

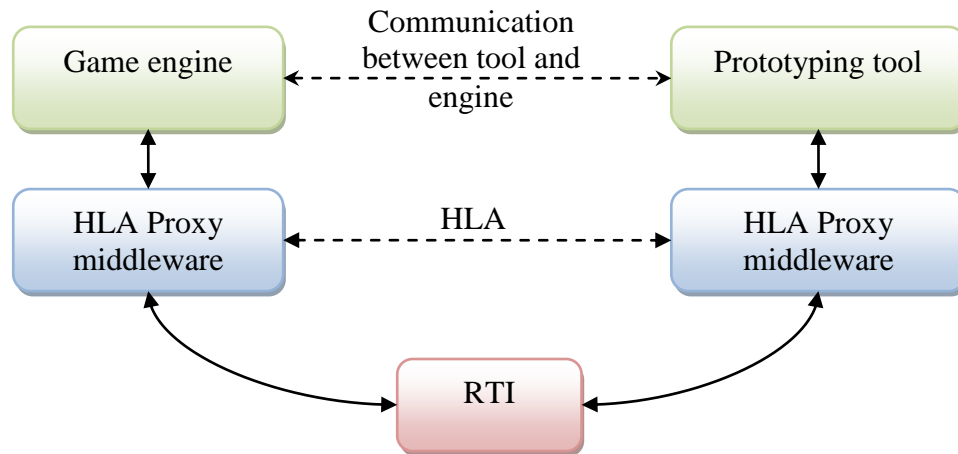


Figure 3-1 Communication between engine and prototyping tool

The Figure 3-1 shows the standard use case for communication of prototyping tools and game engines (marked as interleaved arrow). It does not matter how sophisticated the interaction between game engine and prototyping tool is. From the low level point of view it is just simple data exchange between both sides. From the high level view both sides can be taken as two simulations. Game engine simulates

the synthetic world and artificial beings' bodies. A Prototyping tools simulate the artificial beings' brain.

The Figure 3-1 also shows the main idea of this thesis. It might not be possible to directly incorporate HLA support to an existing application directly. The application might not be able to implement required interfaces by limitations in its design. More common problem is that an application does not provide source code for all of its parts and therefore its modification is not possible and must be done only within range of accessible source code. Therefore additional layer, the HLA Proxy middleware, has been introduced. The goal of the HLA Proxy middleware is to provide all required functionality for enabling HLA support in game engine or prototyping tool (in fact in any 3rd party application).

The agent based platform and prototyping tool forms a HLA federation. Therefore they must define FOM that will be used for their communication. It is out of scope of this thesis to provide full FOM definition. The thesis must provide flexible solution which can adapt FOM changes easily. The aim is to implement support for automatic source code generation for the HLA Proxy middleware. This way it is possible to easily modify the communicational protocol between an agent based development platform and a prototyping tool by simple change of FOM document and rebuilding of middleware.

Object instances are going to be created and published by game engine only, therefore there is no need to use ownership transfers and other advanced HLA features. A prototyping tool is going to communicate only by sending interactions back to an engine. In fact there is no difference in modeling of entities by interaction or attribute updates. Both approaches are equal. Interactions can be used for modeling of attribute updates and attribute updates can be used for signaling of interactions. The decision to use the interactions is just much more intuitive.

3.1 Synchronization

A game engine must be synchronized with the prototyping tool even when the tool is not sending any interactions/events. It is possible that the game engine can produce too much entities updates and overload the prototyping tool. The tool then starts falling behind and will not be capable of performing of any reasonable interaction.

For the real-time synchronization both of them must be running in time regulating and time constrained mode. Unfortunately such high speed synchronization can destroy performance of the game engine. This is a limitation, but it is better to have a working and slow than fast and out of sync solution. The both an agent based development platform and the prototyping tool have to use time regulating and time constrained management. This type of synchronization imposes new requirement for a prototyping tool that it must be regularly advancing its time (even in cases when it is not interested in time synchronization at all). Otherwise the game engine will lag waiting for time advance acknowledgment.

The last requirement is to select how both federates (game engine and prototyping tool) will see the logical time. As they both are designed to work in iterations of some internal loop (it is not possible to design them in different way), the obvious solution is to use discreet time axis.

This assumption can lead to complications, due to the fact that today's engines are designed to try to use all available resources as much as possible. If the game engine overloads the host machine its performance will drop, thus the engine

starts running its simulation on lower frame rates (typically the rendering frames per second are not related to simulation frames per second). The engine usually passes explicit values describing how much time has passed since last update to its internal loops. This can be used directly in time advancing mechanisms of the HLA. To allow support for such irregular updates the time must be represented by continuous function.

The thesis aim is to create a prototype which utilizes time representation as a continuous function, but it advances time by 1.0 during every game engine's frame. The reason is that the HLA Proxy middleware is designed to be usable in different applications that cannot provide any time information. The result of specific 1.0 increment is that game engine's federate representation logical time will not match the internal engine logical time. This does not limit the usage of the HLA Proxy middleware in any way and specific time support can be added when needed.

4. Analysis of HLAProxy architecture

The architecture of the HLAProxy middleware must be simple and flexible. It must be flexible enough to allow easy integration to most applications. The HLA-Proxy is going to be implemented in C++ language. There is no other option, because the C++ language is used in today's game engines (agent based development platforms).

The middleware is going to be a cross platform solution therefore it is going to use some portable libraries, which works on both Windows and Linux operating systems:

- Boost libraries[11] – usefull library for general purpose C++ development
- Intel Thread Building Blocks[12] – library containing various multi-threaded tools.
- Libxml 2[13] – Cross platform XML parses

The Figure 4-1 shows the proposed architecture. The application is always using Common Interface to perform any of the HLA related tasks (e.g. startup or shut-down). The application can register handlers through the Common Interface. These handlers will be invoked when a registered event occurs. The modules can communicate with each other, but never directly they must use the Common Interface. There are some exceptions done for performance reasons. Detailed explanation is mentioned in chapter 5.1.

The RTI Communication Module provides communication through the RTI. It transparently handles connecting, disconnecting, sending updates and receiving of updates. The RTI support module provides supporting tools like value encoders, value decoders and management of object instances. The RTI Support Module also contains mechanism for storage of values received from the RTI. Application Specific Support Module can be anything that application might require to be present for successful integration of HLA related features.

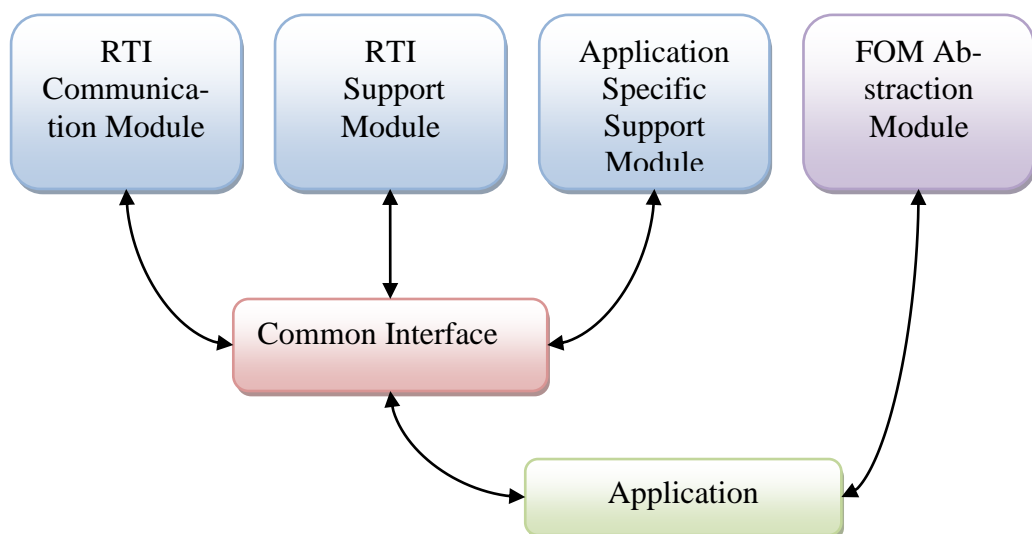


Figure 4-1 Architecture of HLA Proxy middleware

The FOM abstraction module provides user friendly access to object classes and interaction classes defined in the FOM document. The module is the only exception in the architecture's approach to route all communication over the Common Interface.

4.1.1 Common Interface

It is not possible to define Common Interface as a set of functions or class methods in respect to variability of applications. Every application has some specific needs and therefore it will require different tools in Application specific support module (e.g. different types of loggers or configuration management). It can be done for the RTI related parts, because they are designed for specific purpose and therefore their interface is well known.

The result is that Common Interface is going to contain fixed set of functions and methods, but has to be modular too. The modular part of the interface can be designed as a two tier architecture. The first part is fixed and well known and allows obtaining of other specific interfaces on demand via plug-ins.

The detailed description of design of common interface can be found in the chapter 5.1 because its definition is closely related to implementation of the proposed architecture.

4.1.2 The RTI Communication Module

The today's world of HLA consists of multiple version of the standard. Possible choices for RTI communication module are HLA1.3, IEEE 1516-2000 and the IEEE 1516-2010 (HLA Evolved). The difference between HLA 1.3 or IEEE 1516 and the latest version are significant caused by many years of development. The HLA Evolved contains significant changes, like:

- Support for WSDL federates
- Dynamic Linking Compatibility between multiple RTI implementations.
- Encoding helpers

The Web Service Definition Language (WSDL) support allows connecting to the RTI through web services interface. The Dynamic Linking Compatibility solves problem with switching to another RTI implementation. The older standard did not declare any rules and therefore the change from one RTI implementation to another was not straightforward. The new design guarantees that an application, which is not using any RTI private API, can switch to another RTI only by using different shared library. Encoding Helpers are defined as an API for values decoding/encoding between the format that is transferred by the RTI and local representation. Every RTI that supports latest HLA standard will provide these tools.

Older federates can still communicate with federates that are using HLA Evolved standard. The only requirement is to run the middleware with HLA Evolved compatible RTI middleware.

The thesis is the initial phase that is supposed to deliver proof of concept of the main idea. This can be the start of large project in the future. This is a good reason to comply with the latest version of the IEEE 1516-2010 standard as it may allow

better compatibility and better interoperability with other federates in future. The IEEE 1516-2010 standard is very complex. Providing full support for all of its features is out of scope of this thesis.

4.1.3 The RTI Support Module

As was mentioned earlier the IEEE 1516.1-2010[8] standard defines encoding of values transferred through a RTI, but not for all types of values. The federate developers can design their own encoding for any arbitrary type as long as all federates participating in a federation execution use same technique for encoding (marshaling) and decoding (unmarshaling) of the transferred values. The IEEE 1516-2010 provides encoding helpers for following types:

- Simple types
- Fixed records
- Fixed arrays
- Variable arrays

The encoding helpers provided by the RTI might not be suitable for every project and their usage is optional. The federate developers can implement their own encoding helpers (for example to obtain higher performance). Such helpers should be provided as a part of the RTI support module.

The IEEE 1516.1-2010[8] standard does not define any technique for storing the federation data. Many vendors, which are delivering HLA based solutions, provide data storage inside their products to ease the developers work. The RTI support module has to do the same.

Selection of correct data storage solution is not an easy task. The HLA defines federation execution save/restore mechanism. Briefly described it a technique how can be all federates notified to save or restore its state. This can be used to save the state of whole federation execution and terminating whole complex simulation without losing its progress. The federation execution can be restored even when its composition of federates has changed (the HLA defines exact rules for saving/restoring a federation). The middleware is not going to implement federation save/restore mechanisms in this initial stage of development. It is a very nice feature and it might come handy in agent based development platforms, but not all platforms can support it due to limitation of its design and implementation (for example, game engine usually cannot save its state when it is a multiplayer game). By not supporting save/restore the middleware does not need to support persistent storage of data attributes. But it should be designed to allow addition of persistent storage support when needed. The selection can be narrowed to in-memory databases only. Currently there are only two options to start with.

1. Utilize some existing in-memory database solution.
2. Provide custom in-memory database implementation.

Usage of existing in-memory databases seems tempting, but has many limitations. Many of available databases are designed to support SQL language. This is a feature that is not required by the middleware, and SQL support costs some performance. Also the SQL based approach puts some limitation on the design of the middleware as it has to be SQL aware. The customized database implementation requires more work to be done, but it is going to fulfill the requirements completely.

It is hard to select a correct option, because the real requirements do not come from the middleware itself, but from the deployment scenarios. There are two extremes that might occur:

1. Integration of the middleware into an application running on common desktop machine (e.g. utilizing a game engine).
2. Integration of the middleware into an application running on an enterprise grade server (e.g. dedicated for data processing only).

The requirements for the first scenario are mainly to

- a) have a database with reasonable memory footprint and
- b) have effective write/read operation performance.

The middleware shares resources with other applications (for example simulation platform running on the same machine etc.). In the end the client machine does not need to utilize database at all and data can be stored directly inside the database of simulation platform. Sharing data with simulation platform's internal database is usually not possible. A simulation platform does not know about the clients and can destroy data without providing any notification. Also there might be problem to solve thread synchronization issues, because a simulation platform may have some specific private rules for accessing the database.

The second scenario is completely opposite. The middleware is used to concentrate data from all federates to one large database and then respond to client queries. In this scenario, the main requirement is to fully utilize available resources (e.g. many threads and memory) and to answer the queries as fast as possible.

The result of our investigation is to use customized implementation of the database, for following reasons:

- Customized database can be designed precisely to meet specific requirements, which are yet unknown and may arise during development.
- Customized database can serve as a proxy to some other database (e.g. SQL like database). It can be extended to support persistent storage with guarantee of proper ordering of operations.
- The database scheduling algorithms are under our middleware's control and some performance and correctness guarantee can be given.

The scheduling algorithm is the most important part of custom database implementation. The database must reflect all write/read operations in an order received from HLA to provide consistent data storage and retrieval. All time stamped operations are strictly ordered and scheduler must be capable of keeping the ordering. This is a problem with transactional databases, because they are usually able to restart or rollback running transactions and restart them again. Such reordering can result in different order of operations resulting in inconsistent view of the world state.

If the database is modified to act as a proxy for some real database then the scheduler can be still used to keep the ordering of operations. This design allows usage of any arbitrary real database that is not capable of keeping strictly ordered operations. The result is that the custom implementation of database is the most reasonable decision for the prototype implementation.

4.2 FOM Abstraction Module

The FOM Abstraction Module is exception and application can access it directly. The main goal of the module is to provide native C++ interface for classes and interactions defined in FOM document. The object classes defined in the FOM Abstraction Module access the database in RTI Support Module transparently every time a new value of class instance attribute is being read or written.

The FOM module is separated from others because it is going to change every time the FOM changes. Also an application is the main consumer of its data classes and therefore it needs fast access for performance reasons.

There is no benefit of having more abstract data model design, because every federate is designed to support selected object classes and interactions. Every change of FOM that is not backwards compatible requires modification of an application. Therefore the FOM abstraction module and application can be tightly coupled.

This does not apply to other modules as they can change and improve without the need to modify the application itself.

The FOM Abstraction Module is automatically generated to match selected FOM document. This design allows fast adaption of new FOM document by the middleware.

4.3 Typical use cases

The HLA defines very complex interface between the application and the RTI. The developers have to implement a lot of code to handle all service invocations correctly. The desired solution is to provide more simplified interface that allows easy adaptation of HLA to existing or newly developed application.

Startup and shutdown shall be handled by simple set of methods that can obtain all required parameters from middleware's configuration (through Application Specific Support Module) or by simple invocation of Common Interface's interface methods.

Access to interaction and object classes shall be simple, straightforward and using simple interface with C++ native types. The user does not need to see any internal details of HLA implementation. The Example 4-1 shows use case for accessing the HLA data classes inside FOM abstraction module. The example is written in pseudo code.

```
// Create and publish new instance through HLA
Instance = Create_hla_instance("HLA_object_class",
                               "HLA_instance_name");

Instance.setHealth(100.0);
Instance.publishValues();
delete Instance;

// Obtain latest known values of attributes
Instance2 = get_hla_known_instance("HLA_instance_name");
health = Instance2.getHealth();
delete Instance2;
```

Example 4-1 Data manipulation use case

By this way user can easily manipulate with HLA entities without seeing any background synchronization and data encoding and decoding from network format.

The second required option is easy registration of handlers for specific HLA events (for example interaction receiving). The desired use case can register callbacks directly to every significant event. The pseudo code is shown on following example.

```
// Interaction handler
handleInteraction(Specific interaction)
{
    interaction.getHealth();
}

// Registration of interaction handler
registerInteractioHandler("HLA_interaction_name",
                        handleInteraction);

// Sending of an interaction
SpecificInteraction i(health = 100.0);
sendInteraction(i);
```

Example 4-2 Event handling

Another use case is targeting processing of long term calculations over data without blocking the processing of other updates or exclusive access to an instance that allows access to its members only to its owner.

```
// Interaction handler that modifies selected object instances
handleInteraction(Specific interaction)
{
    Instance = getInstance("HLA_instance_name");
    Instance.lock();
    ... perform some exclusive operation on the data
    Instance.unlock();
}

// Interaction handler executed in separate thread that does not
// block other data processing
handleInteraction(Specific interaction)
{
    Instance = getInstance("HLA_instance_name");
    InstanceReadOnly = Instance.getReadOnlyCopy();
    ... schedule long computation above read only copy to separate
        thread
}
```

Example 4-3 Time consuming and exclusive operations

These are the most important use cases that need to be very simple and effective. Of course many other similar use cases can be found very easily. There is no reason to try to map whole HLA interface to the pseudo interfaces now. The examples show very easy understandable API that would be easily adopted by developers.

4.4 Selection of the RTI

The most important part of the thesis is the correct selection of a RTI implementation. Obtaining a RTI is not as easy task as it might seem to be. As usual there are two categories of the RTI implementations:

- Open Source implementations
- Commercial solutions

An open source RTI is the most desired product, because it can provide all of the required features for free. Unfortunately there is not so much open-source implementations and most of them are not finished yet or do not comply with the IEEE 1516.1-2010[8] standard. Second problem is that usually the open-source RTIs provide Java interfaces only and therefore cannot be used for implementation of this thesis. Following RTIs are freely available under open source licenses:

1. OpenHLA
2. CERTI
3. Portico RTI

The Open HLA provides IEEE 1516-2010 support and also contains tools from automatic source code generation from FOM definition. Unfortunately the latest available version 0.5 is lacking C++ interfaces.

The CERTI has currently only a partial implementation of IEEE 1516-2010 C++ API and it will take some time until it is finished.

Portico RTI is really promising implementation and continuation of project known as jaRTI. Currently Portico is lacking support for IEEE 1516 and IEEE 1516-2010. It does not provide C++ interfaces yet.

The overall result is that open-source RTIs are not ready to be used in this thesis.

The second group of the RTIs is standard commercial products. They are far better in standard compliance. Many of them are certified by the Department of Military Simulation (that means they are verified to support the HLA standards). However it is not an easy task to obtain free evaluation of the RTI implementation. The possible choices for the thesis were:

1. Pitch RTI
2. MAK RTI
3. RTI-NG Pro

The Pitch company does not provide evaluation version of the RTI. They provide only a very limited RTI demonstration. The RTI, which is limited to an execution time of 10 minutes and maximum of 100 data reflections. This is not acceptable limitation for the development of the middleware and its testing.

The MAK RTI is available as an evaluation version, which can run for unlimited time. Unfortunately the RTI allows only two federates to be joined in one federation at the same time. This is a limitation of scalability, development and verification of complex federations consisting of many federates. This is not limitation the middleware in its initial phase of development..

RTI-NG Pro is product of Raytheon company. The RTI can be obtained as an evaluation version running for 30 days with no restriction or limitation of its features.

All commercial RTIs share the common features:

- High performance (up to 100000 updates per second)
- Support for multiple operating systems
- C++ API

- Verified by U.S. Department of Defense's Defense Modeling and Simulation Office for at least IEEE 1516 compatibility
- Have advanced implementation targeted on performance and usability (e.g. support for large WAN deployments)
- Various tools for observing state of federations and its federates
- Commercial support

The MAK RTI was selected as the RTI that will be used in this thesis. The reason is that a free evaluation version is not limited in time and therefore can be used during whole implementation, not only for 30 days. Other limitations of evaluation versions are not important for this thesis.

5. HLAProxy internals

This chapter discusses the architecture of the middleware at the implementation level. The discussion contains also discussion of unsuccessful decisions that has been made during development.

The implementation contains more features that are beyond scope of analysis of architecture in chapter 4. Some of them were specifically designed for future continuation based on this thesis.

This chapter also does not define exactly all interfaces. The main idea is to discuss ale implementation related details, but keep the explanation simple. The full detailed description of each interface can be found in the HLA Proxy middleware developer documentation, which is provided on DVD that is attached to this thesis.

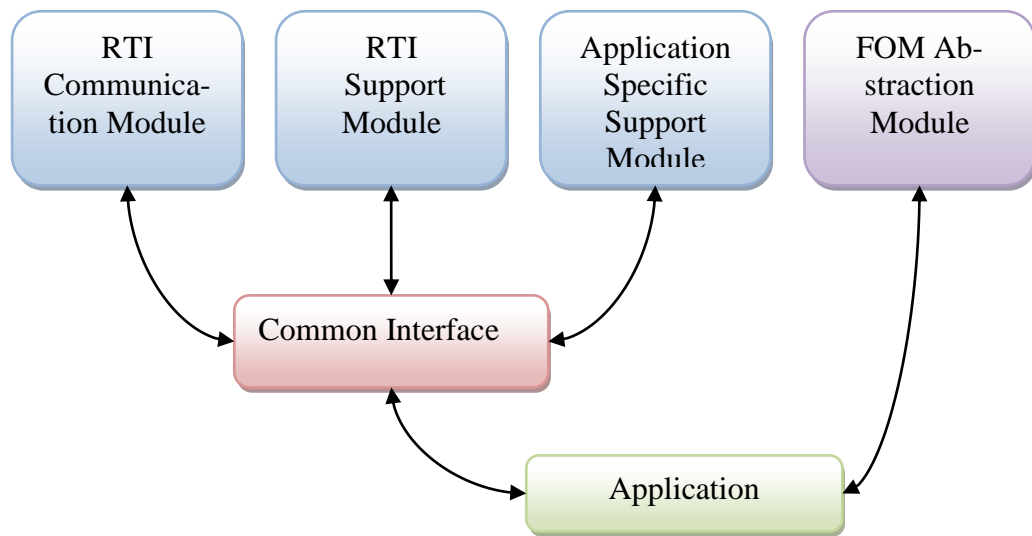


Figure 5-1 Architecture of HLA Proxy middleware

For convenient reading the Figure 5-1 contains resulting architecture from chapter 4.

5.1 *Common Interface*

The definition of Common Interface in chapter 4 is left intentionally too open. It is not possible to define final interface during analysis stage as the interface contains modular parts. Before defining the common interface it is required to understand the architecture of the HLA Proxy middleware from the implementation point of view.

The HLA Proxy middleware is designed as a modular architecture. This approach allows easy modification and addition of new features in the future. The modular architecture of the middleware can be seen on the Figure 5-2.

The middleware is composed from the three layers:

- Plug-in management
- Plug-ins
- Components

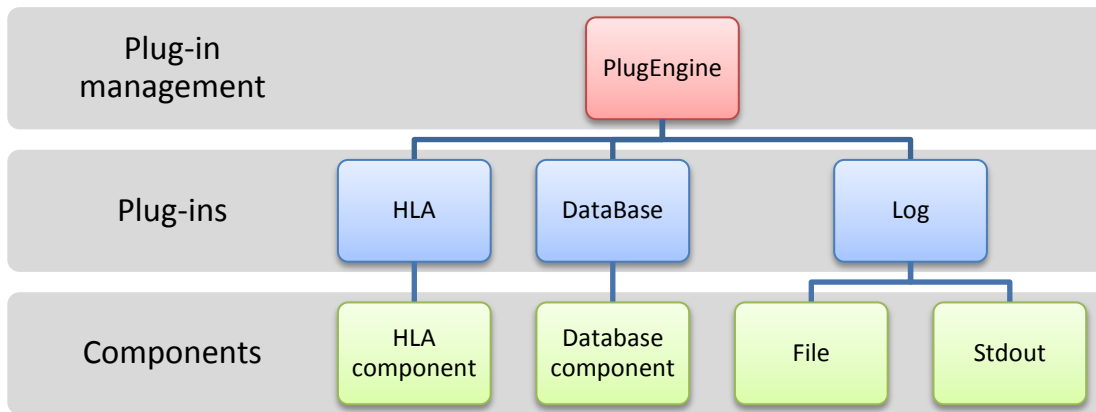


Figure 5-2 Modular architecture

The plug-in management (realized by *PluginManager*) part of the middleware is the key part that must be used by every application that wants to utilize the middleware. The plug-in management API is the fixed part of the common interface and provides interface for loading additional plug-ins and creation of instances of components that are stored inside plug-ins. Components can be freely stored in any plug-in. The middleware does not impose any restriction for component placement.

A component is a piece of the middleware that performs some activity. The component can be either a single class of programming language, or whole subsystem composed of multiple classes and threads of execution. The component can also utilize other components to perform its activity.

Communication with a component is realized through the component interface. All defined component interfaces form the variable part of the Common Interface definition. Component interfaces are accessible by including publicly provided header files. The suggested naming convention for component interfaces is to insert 'I' before component's name (e.g. component with name HLA has interface with name IHLA). Components can share the same interface specification, for example the configuration component described in chapter 5.6.1. Usage of the component interfaces allows treating all components with same interface as equal.

The HLA Proxy middleware can easily modify its behavior, by instantiating another component that implements same interface. Other benefit of using component based approach is that existing source code does not need to be modified as long as the used component's interface does not change. The source code remains compatible with all new components that fulfill the selected interface specification.

5.1.1 Plug-in internal structure

Plug-ins are implemented as shared libraries. Every plug-in has same internal structure and must export specific plug-in public interface from its shared library. The plug-in contains following parts:

- Plug-in description
- Component factory

Plug-in description is simple structure that can be accessed from the outside world. The structure contains some descriptions about the plug-in:

- Author's name
- Plug-in name

- PlugEngine version
- List of all required plug-ins that must be loaded before.

The plug-in interface contains only simple C code. Exporting of C++ classes and STL containers from shared library is problematic. All various possible problems are described in Appendix D.

Stored components inside the plug-in cannot be accessed directly. The plug-in must provide component instantiation factory. The factory must be able to return total count and names of components it can instantiate. The factory must also be able to create new instance of the component when requested. This allows creation of new instances every time the *PluginManager* is asked to, or returning of a singleton instance of a component. Further specific instantiation techniques can be added without modification of existing application, because the mechanism of component instantiation is hidden behind factory interface, which does not change.

5.1.2 Component registration and instantiation

Every application that uses the middleware must link and create instance of *PluginManager* (obtain the implementation of fixed part of the Common Interface). The component instantiation process consists of following steps:

1. Load of external plug-in
2. Load of additional necessary plug-ins
3. Registration of all component factories contained in loaded plug-ins.
4. Instantiation of the selected component in appropriate factory

The first step is initiated from the application and it simply tells the *PluginManager* to load a shared library. At this step the *PluginManager* tries to load the shared library and verifies if it is a valid plug-in. Every valid plug-in must export plug-in interface from its shared library. If the plug-in is valid the next check is whether the plug-in has been build for correct version of the *PluginManager* to avoid binary incompatibility. When all tests pass the plug-in loading continues with next step.

During second step the *PluginManager* reads plug-in information section and identifies all required plug-ins that must be loaded. The list of additional plug-ins contains shared library names. If the plug-in is compatible with Windows and Linux based systems it must correctly fill in shared libraries names according to the operating system naming conventions. It is up to the plug-in developer to provide correct and complete list of all required plug-ins. The plug-in dependencies can form a cycle in their dependency graph it is not a problem for *PluginManager* to solve such dependencies. Every dependency is loaded by following above mentioned steps.

When the 1st and 2nd step has been successful the *PluginManager* registers all component factories that are inside loaded plug-ins. This step registers component names and their associated factory pointer to the global component map in *PluginManager*.

The fourth and last step is to create interface requested by an application. The *PluginManager* tries to find component instance name in the global component map. If the search is successful then the associated factory is used to create instance of the specified component.

If something goes wrong the exception is thrown from the *PluginManager* instance and all temporary data are automatically discarded.

5.2 RTI Communication Module

The RTI Communication Module is realized by single component called HLA. The component is instantiated as a singleton instance. Therefore it is not possible to have multiple connections to the RTI at the same time. Even though the IEEE 1516-2010.1 states that it shall be possible. Participation in more than one federation has no benefit for this thesis.

The HLA networking component handles all RTI communication. The HLA component is not only a networking proxy. It is a full federate implementation. The implementation is done according to the IEEE 1516-2010.1[8] and is compatible with the MAK RTI 4.0.4 and newer. It should allow replacement of the RTI thanks to the Dynamic Link Compatible interfaces introduced in the latest version of HLA standard. But it has not been tested with different RTI implementations.

The implementation does not offer full support for all HLA defined services. Currently supported services are:

- Simple advance time support
- Publish interaction class
- Subscribe interaction class
- Send interaction with/without TS
- Receive interaction
- Publish class attributes
- Subscribe class attributes
- Reflect attribute updates
- Update class attribute values
- Join and resign federation execution

Not all features of above mentioned services are supported. For example the HLA component automatically subscribes/publishes all class attributes. It is not possible to publish or subscribe only subset of the class attributes. The same applies for updates too. The limitation is intentional as it provides simpler interface for application developer and lack of such features is not limiting in usage of the HLA Proxy middleware in this thesis.

All components' methods catch the RTI exceptions and provide simpler interface that only return true or false depending on whether the call of method has been successful or not. The reason is to not throw exception across shared library boundaries unless it is absolutely necessary. Error conditions are reported through *IContectLog* interface (explained later).

The HLA component provides automatic support for encoding, decoding of values received from or send through the RTI. The HLA transports data in a format that is defined in the FOM document. The details about automatic encoding and decoding support are explained later in chapter 5.4

5.2.1 HLA Federate and its configuration

The component that is responsible for configuration management is described in 5.6.1. This chapter describes configuration properties that must be present for using the HLA component. All properties are described in following table.

| Property | Description |
|----------------------------|--|
| FederationExecution | <i>The name of federation execution to which the federate should connect.</i> |
| FederateType | <i>The type of the federate</i> |
| FederateName | <i>The name of the federate</i> |
| FDD | <i>The FOM document file</i> |
| Log | <i>Target log output context</i> |
| SyncMaster | <i>True when the federate has master role in federation synchronization during startup</i> |
| FederatesCount | <i>The total amount of federates that will participate in federation execution</i> |

Table 5-1 HLA component configuration

It is not possible to configure time management through configuration file. Usually a federate is designed to support some time management scheme and it can't be simply switched to different one just by a simple change in HLA component configuration. The time regulating/constrained configuration has to be done directly by developer through *IHLA* interface. All properties must be present in configuration file otherwise the HLA component won't start communication with the RTI

The HLA Proxy middleware performs synchronization of all federates during its startup. The startup synchronization is done according to following steps:

- The SyncMaster federate waits until the total amount of federates is equal to FederatesCount. Then it announces new synchronization point. The normal federates join in and start waiting for synchronization point announcement.
- Every federate acknowledges that they have reached the synchronization point and wait until it receive notification that federation is synchronized (from the RTI).
- All federates are now synchronized.

This is a very simple synchronization protocol, but it is enough. It is not possible to join the federation lately. Such synchronization would require much more complex design, because lately joined federates might receive data updates when they are not ready to process them. Also lately joined federates have to synchronize its logical clock if they are time regulating and time constrained.

The HLA component must obtain pointer to the database component's interface *IHLADatabase* for successful startup.

5.2.2 Object class support

The HLA component provides main interface for manipulation with object classes. The interface has following methods:

- createInstance
- getInstance
- deleteInstance
- updateInstance
- registerClassCallback
- publishClass

- `subscribeClass`
- `registerInteractionCallback`
- `publishInteraction`
- `subscribeInteraction`

Implementations of methods are straightforward by calling RTI services according to the IEEE 1516.1-2010[8] standard.

The second set of methods is not publicly exposed and takes care of instance discovery, receiving of attribute updates. The HLA component directly uses the *IHLADatabase* interface (which was obtained at startup of the HLA component) for storing the values. The database component is described in the RTI support module implementation in chapter 5.3.

Object class instance data and handles are stored in handle caches. Currently the cache implementation is present in the FOM Abstraction Module. Encoding and decoding is performed automatically when needed. Encoder and decoder methods are closely related to the data model defined in the FOM document and handle caches. Therefore object methods accesses the FOM Abstraction Module directly by private interface. This decision keeps all related code that depends on FOM in a single place, but also improves performance by using of direct private interface.

Currently the HLA component does not support ownership transfers. Ownership transfers cannot be implemented at this level. If one federate asks another for ownership transfer for a specific attribute then the currently owning federate can decide whether it will or won't transfer ownership of an attribute to the requesting federate. This is something that should be decided by the application and not by the middleware itself.

The support for future addition of ownership transfers is present. The application cannot write into non-owned attributes and non-owned attributes are not published during call to *updateInstance*. The thesis does not need to solve ownership transfers, because the problem that is being solved does not need ownership transfers at all.

5.2.3 Interactions

Interactions cannot be handled by the middleware. It is up to the application to process them. The HLA component provides interaction callbacks that can be used for interaction handling. The callback is called from within RTI's thread, so the interaction handlers should be small and simple functions or the performance of the middleware will drop. Not only the performance of middleware may suffer, but also performance of an application that is using the middleware. The reason is that *evokeMultipleCallbacks* services are usually called from within per-frame loops that are very sensitive on performance.

It does not mean that complex interaction handling is not supported. The callback may for example decide to create a database snapshot and then start complex calculation in a separate thread. This allows user to give control back to the RTI and perform operations in background. The database is designed to support concurrent access to its data so there should not be a problem with long updates, when proper record locking is in place. These techniques are illustrated in use case analysis in the chapter 4.3.

The HLA plug-in supports one callback per interaction type. User has to register his callback by *registerInteractionCallback* method of the IHLA interface. No

interaction callbacks are received until the user subscribes to receive selected HLA interaction class. The interaction class is passed to the callback by reference. The class is instantiated and destructed automatically upon returning from a callback call. If there is need to maintain interaction parameters' values longer the user must create his copy of parameters. The example of the interaction callback is shown below.

```

Void handleConsoleCommand(BaseInteraction &interaction)
{
    ConsoleCommand &cmd = static_cast<ConsoleCommand
&>(interaction);
    std::string cmd_str = cmd.getCommand();

    cout << "Received command: " << cmd_str << endl;
}

```

Example 5-1 Interaction handler example

Sending of an interaction is done in the same way. Interactions are not meant to be allocated dynamically. The only one supported method of interaction sending is shown on following example.

```

Std::string cmd;
ConsoleCommand conCmd;

conCmd.setCommand(cmd);
pHLA->sendInteraction(conCmd);

```

Example 5-2 Interaction sending example

Interactions can be sent also with timestamp. The user can select any time that is larger than current federate time plus its lookahead. This allows scheduling of interactions deliveries to other federates in future.

The reason why all arguments are being passed by the reference is simple. The interaction does not need to be kept in memory longer then its associated callback finishes its work. Usage of references does not tempt application developers to store the pointer for future access to interaction parameters.

5.3 RTI Support Module

The RTI Support Module currently contains only a database to store received attribute values. The database plug-in is one of the core parts of the HLA Proxy middleware. The plug-in contains only one component called *HLADatabase*. The *IHLADatabase* represents component's Common Interface. The database is provided as a separate plug-in, because the future replacement of database implementation might be required.

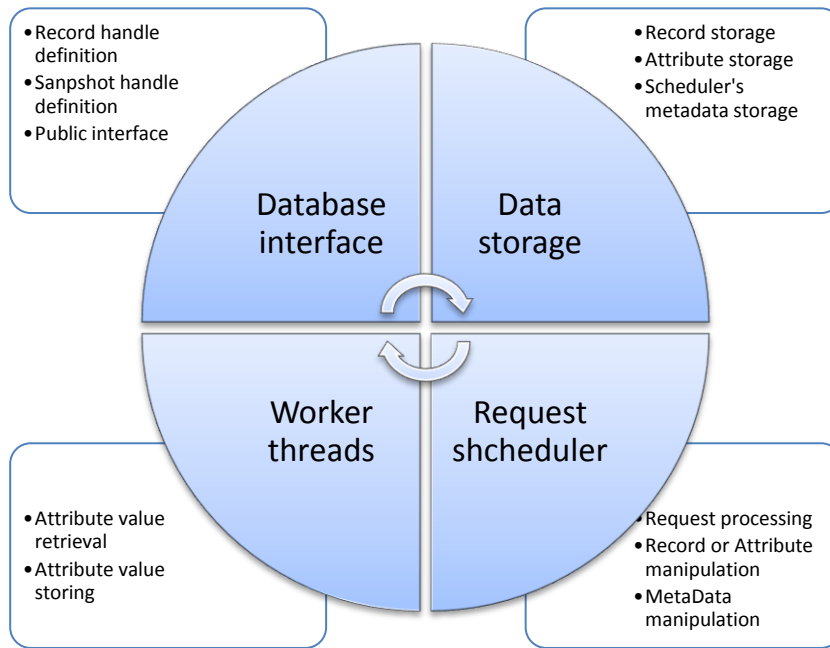


Figure 5-3 Parts of the database component

The basic structure of the database component can be seen on Figure 5-3. The database component is divided into four main parts that cooperate together:

- *Database interface* –Provides implementation of database's Common Interface for database management and record access by application
- *Data storage* –Provides mechanism for storing values. Currently the implementation is designed for in-memory storage only.
- *Database scheduler* – The main responsibility is to process request in a strict order according to the order they were submitted through database interface. The dispatcher also performs some data manipulation directly.
- *Worker threads* – Perform attribute's value storing and retrieval.

There is no need to access database directly if the application is using HLA component for real-time synchronization. The HLA is going to access the database component directly when needed.

The Database component does not know anything about HLA. If an application accesses the database data directly then it is possible that it can perform operations that are not correct according to the HLA rules. Operations performed by direct access to the database interface from an application must be done very carefully to not interfere with HLA processing that is might be happening at the same time (e.g do not change ownership of attribute).

The design of the database allows concurrent access to its data records from multiple threads. Therefore it is safe to use the database interface from multi-threaded context. The resulting order of data access will be the same in which the threads registered its requests. Therefore it may be needed to introduce additional synchronization at application level to avoid race conditions.

The database scheduler handles every database request for accessing stored values. The scheduler is running in single thread and keeps strict ordering of operations according to time when they were submitted. Some operations with data are

processed directly by the scheduler. All read and write operations are dispatched to worker threads and are performed asynchronously and in parallel if possible. If there is possibility of collision the scheduler postpones request that is currently being processed until it is safe to execute it.

The data storage is currently designed for in-memory database, but it can be extended to support different kind of storage in future. The storage part must be always capable of storing scheduler's metadata in memory. The metadata are required for correct order of request processing and must be present in memory for performance reasons (the metadata structures are explained in chapter 5.3.2.2).

5.3.1 Database interface

The database interface does not perform any complex tasks. The main purpose is to enable application or database component client to perform following tasks:

- Start and stop the database
- Insert or delete a record
- Set or get record attribute values
- Create a database snapshot (the implementation is not finished for more details see chapter 5.3.3.3)

The implementation of the interface is straightforward. Usually call to the interface method ends in submitting of new request to the scheduler. Then according to the request type the control is immediately returned back to the caller or the caller is blocked until the request processing is finished. More detailed description of request processing is explained in chapter 5.3.2.

The database interface also defines *IRecordHandle* and *ISnapshotHandle*. Both handles are designed to be used by application or database component clients. More detailed description is chapter 5.3.3, because understanding of record handles requires knowledge of data storage implementation.

5.3.2 Request scheduler

The scheduler is designed to process all requests in order in which they were submitted. This is important for HLA as it guarantees that all values stored in the database are stored in same order as they were received. It allows for example accessing objects during interaction processing, because the interaction can read all updated attributes that were received before the interaction.

The aim of the scheduler is to fulfill following requirements:

- Execute read and write operations in separate threads.
- Keep data in consistent state
- Do not break the time ordering of the events

The requirements are almost the same as for the regular relational database scheduler. There are multiple papers about scheduling of transactions in today's relational databases. The idea presented in this thesis is based on information from the Database Systems: The Complete Book[14]. The main difference between regular scheduler and HLA Proxy's scheduler is that we cannot restart or abort transaction. Aborting or restarting of transaction is not possible, because reordering of operations

is not allowed. Some of the events are received strictly ordered according to the preferred transportation order, which is defined in the FOM document, therefore it is not possible to replay them later. The result of such reordering is that the database will end in out-of sync state.

Lot of time spent on this thesis was by investigating of all possible implementations and designs of the scheduler. Today's database and transactional systems use following mechanisms for performing of data operations:

- *Locking* – This approach requires a lock to be held before an operation executes. There are no conflicts at all, but can lead to degraded performance when accessing same attributes. Also obtaining a lock may involve additional actions to be performed so more CPU cycles will be spent on locking/unlocking operations. The main disadvantage is possibility of dead-lock, which can be prevented by some deadlock avoidance mechanism, or strict locking policy.
- *Timestamps* – This approach does not require locking of data structures. Operations above data are scheduled according time-stamps of the transactions. This mechanism can lead to aborting or rollback of a transaction that created a conflict.
- *Multi-Version* – The data keep history of its previous values, thus it is possible to execute operation that accesses older version and write new version simultaneously. This solution is not bullet proof too, there can be conflict between two operations and can result in abort/rollback of transaction.

The HLA Proxy scheduler's requirements are going against each other. For example, to be able to support multiple threads of executions requires high isolation of transactions so they do not interfere with each other. On the other hand the data consistency, time ordering and never abort or rollback requirements require serial processing of operations.

The result is that there is no clear solution to this problem and none of the today's used mechanism is suitable for this situation. The HLA Proxy's scheduler utilizes combination of all of the approaches to fulfill its requirements. Every request is assigned unique timestamp in increasing order. The timestamp is used to decide which operation is older and therefore has higher priority.

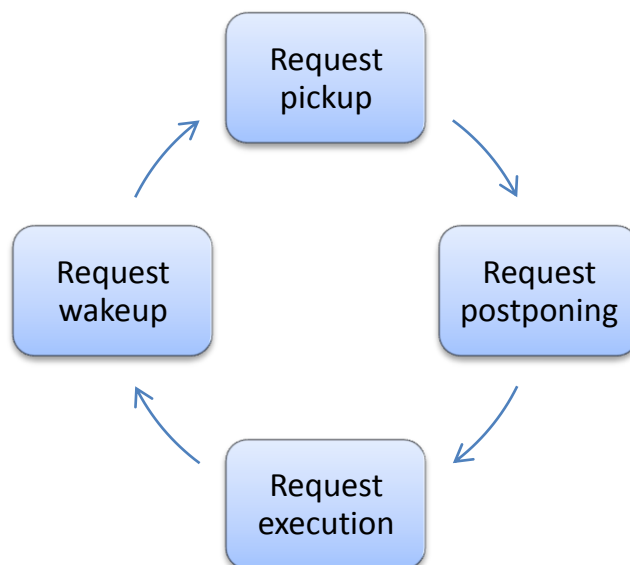


Figure 5-4 Request dispatching

The scheduler is designed to allow concurrent access to different attributes. All operations targeting the same attribute are processed in serial order. The only exception is execution of read operations. Because read operations does not modify the database they are scheduled in parallel.

The scheduler is dispatching all requests in a single thread. The thread is called the *dispatcher thread*. Scheduler contains two structures for incoming requests. One is a standard FIFO queue for new incoming requests. The second is a heap of old requests that are waiting for processing. All access to waiting queues have to be done with locked queue mutex (this applies also for worker threads). For more details about thread synchronization please see chapter 5.3.4.2.

The dispatcher thread performs three simple steps as shown on Figure 5-4. The first step is to pickup waiting request if there is any. The dispatcher thread looks into both FIFO queue and heap and pickups request with the lowest timestamp.

The second step is investigation of the request itself. The dispatcher thread identifies target record and attribute and looks into its metadata structure (detail explanation is in chapter 5.3.2.2). If the request cannot be executed then it is postponed by storing it to either target record's delayed queue or target attribute's delayed queue. If the request has been postponed the dispatcher thread starts again by pickup of next waiting request.

When it is possible to execute the request, the dispatcher thread either performs the desired operation by itself or schedules execution of a request to worker threads. This is a performance optimization, because some of the operations result in a modification of record's or attribute's metadata only and scheduler already has pointers to correct structures and has locked mutex which protects delayed queues (see chapter 5.3.2.3 about record locking and unlocking). All read/write operations are scheduled for execution by worker threads.

After execution of a request (by dispatcher thread or by worker thread) the delayed queues in record's metadata or attribute's metadata (depends on type of operation) are checked and some of the waiting requests are merged back to the scheduler's heap (they are unblocked now, because the blocking request has been executed). All waken up requests were waiting for some time they have lower timestamps than new incoming requests and therefore are picked up by dispatcher thread with higher priority. The data storage model that explains request and attribute representation is described in chapter 5.3.3.

The performance results that were collected by simple measurement are presented in chapter 10.

5.3.2.1 Scheduler heaps

Scheduler's heaps are participating during processing of every request. They must be designed with performance in mind. Otherwise the database will suffer from degraded performance when it is overloaded by waiting requests. The second important part is that internal queues must be error prone as much as possible.

The first important optimization is hidden in specific memory management. The scheduler is designed to not allocate any additional memory for request dispatching and execution. This is done by usage of intrusive structures for request management. The intrusive containers are different from standard STL ones, but provide same capabilities. The usual case is that a STL container keeps a copy of inserted object. But this introduces unnecessary allocation of memory and slower operations with the container (due to copying of values). The intrusive container stores

specific value directly. The stored value must contain members that are required by selected type of container. Those members are called hooks.

The *BaseRequest* class (explained later) contains hooks that allow insertion of a request instance into a *treap_set*[10] structure (combination of tree and heap). Every request can be member of only one queue, because it contains only single hook. This technique adds small memory overhead of three pointers to the size of a request class instance.

Scheduler can then move requests between internal queues without the need of additional memory allocation. This approach has these benefits:

- Allocation of memory cannot fail during copying of request
- Intrusive containers have higher performance due to its design

The *treap* structure is used because the requests in a queue must be sorted, the best possible way to achieve this is to use heap. However heaps are designed to typically operate over array (binary heaps). Unfortunately the requests are not present in an array but they are allocated on a heap or stack of a caller. The *treap_set* container cross links inserted requests to form a binary tree but supports heap operations.

The internal tree representation has performance problem, because insert and delete operations are being done in $O(\log N)$ and long queues may have significant impact on performance. The solution that is used in the HLA Proxy middleware is to use hinted insertion. The idea is very simple. If a request is going to be postponed and put in some waiting queue and the queue is not empty then we can expect that all already waiting requests have timestamps lower than current request (they reached the delayed queue before the request). Therefore it is expected that new request is going to be appended at the end of queue. If not then it is not a problem the *treap_set* structure will find correct place for the request in $O(\log N)$, but hinted insertion has amortized performance of $O(1)$. The hinted insertion has amortized performance guaranteed only if the hint points to a location in a tree, where the request is being inserted to. This is the reason of why long waiting queues does not decrease the scheduler's performance (see chapter 10).

5.3.2.2 Metadata structures

The database records metadata members are described in table Table 5-2. The record's metadata structure contains members required for supporting of request locking and unlocking operations (explain in chapter 5.3.2.3).

| Member | Description |
|------------------|---|
| LockOwner | <i>The IRecordHandle that currently owns the lock.</i> |
| LockTS | <i>The timestamp of operation that locked the record.</i> |
| WaitList | <i>The heap containing all waiting requests for lock releasing.</i> |

Table 5-2 Record's metadata structure

The attribute metadata structure contains almost same members as the record's metadata. The ActiveReads and ActiveWrites are atomically incremented and keeps actual count of running operations.

| Member | Description |
|---------------------|---|
| ActiveReads | Number of currently executed read operations |
| ActiveWrites | Number of currently executed write operations |
| Locked | Not used. Designed for garbage collector support that is required for snapshots |
| WaitList | Requests waiting for this attribute until it becomes available. |

Table 5-3 Attribute's metadata

5.3.2.3 Locking and unlocking of database records

The database supports record locking and unlocking. The locking mechanism allows obtaining exclusive access to database's record for an application or a component database client. The locking is done only by modification of record's metadata and therefore is performed by the dispatcher thread directly.

If the record is not locked than scheduler simply stores the lock owner and current lock request's timestamp to the target record's metadata structure. All requests that are targeting a locked record are being postponed and put in the record's metadata *WaitList* heap. The only exceptions are request initiated by the lock owner.

The lock is obtained immediately, however older requests (with lower timestamp than *LockTS*), which are waiting in record's attributes' *WaitList* are being still executed. This is safe, because only requests from lock owner are going to be executed. Because they came after the lock they must have higher timestamp value than pending requests and therefore will be processed in correct order.

The request unlocking works in the same way, but after execution of unlock request the *WaitList* in record's metadata structure is merged back to the scheduler's heap. Therefore all pending requests are rescheduled for processing.

5.3.2.4 Request structure

The database operations require a lot of information, which have to be passed between scheduler and worker threads to perform the operation correctly. It is not a good practice to pass a lot of arguments to a function in any programming language, because the code becomes unreadable and un-maintainable after some time. Also all of the arguments are stored multiple times on the stack. Therefore the operation request is represented by set of classes.

| Member | Description |
|------------------|---|
| TimeStamp | <i>The timestamp of the request assigned by the database scheduler.</i> |
| ReqType | <i>The type of the request.</i> |
| TimeStart | <i>The time when the request has been created.</i> |
| Duration | <i>The duration of the request processing.</i> |
| Argument | <i>The request argument.</i> |

Table 5-4 Member description of BaseRequest class

The requests are passed between functions by pointer to a request class. The *BaseRequest* contains basic common members, which are shared by every request.

The following table provides description of all its members. The *BaseRequest* is inherited from *bs_set_base_hook*<>[10]. This allows storing the requests inside *treap_set* containers.

Base request defines three virtual methods:

- *acknowledge()*, which is called when the request has been successfully processed.
- *reject()*, which is called when the request is considered invalid.
- *fail()*, which is called when there is a critical failure during request processing.

The *BaseRequest* does not perform only calculation of time duration of the request processing. Every inherited class must call overloaded method from *BaseRequest*, otherwise the performance statistics will not be accurate.

The *BaseRequest* also contains Argument attribute that can contain additional value, required during request processing. The Argument is of *boost::any*[10] type and can hold any arbitrary type. The Argument is used mainly for transfer of value during storage and retrieval operation. The class should not be enhanced with more attribute arguments in future. If there is a need to transfer more than single additional value then the most preferred approach is to store multiple values directly into the Argument member by using, for example, *std::pair*. Introduction of new class attribute is going to consume more memory even for requests that don't use the member.

The first type of request that originates from middleware's user is the *ClientRequest*. The *ClientRequest* extends the *BaseRequest* by following members. The *ClientRequest* is used for asynchronous request from the client. When the client submits asynchronous request a new *ClientRequest* instance is allocated. Once the scheduler completes request processing it is automatically destroyed during acknowledge, reject and fail method calls. The asynchronous client request cannot return any value to its initiator. However it can report the error during its execution. Unfortunately there is no request initiator waiting, so the database scheduler has to take care of the error by itself. Currently the error is silently ignored as it can't be decided how much serious it is.

| Member | Description |
|----------------|--|
| Record | Handle of the operation target record. |
| Function | Pointer to the function that shall be executed to process the request. |
| AttributeIndex | Index of the target attribute. |
| ErrorMsg | Error message in case of failure |

Table 5-5 Member description of asynchronous request

The last type of request is the *SyncClientRequest*. The synchronized client request extends standard client request and allows blocking of user call until the request has completed. The Table 5-6 describes synchronized client request members. The request is not self-destructing itself, because the thread, which has initiated the request, is still waiting on its result. Therefore memory cleanup is responsibility of the caller.

| Member | Description |
|------------------|--|
| Result | <i>Enumeration which describes the result of request processing.</i> |
| ResultPtr | <i>Pointer to the result's <code>boost::promise</code></i> |

Table 5-6 Member description of `SyncClientRequest` class

5.3.2.5 Synchronous requests

The database API can work in two modes: synchronous calls or asynchronous calls. The asynchronous calls are non-blocking but it is not possible to determine their state or end of processing. The synchronous calls behave as a blocking API. The asynchronous calls are simple. The *ClientRequest* is allocated, filled in and then send to the dispatching queues. Once the request processing has finished the request is destroyed. The asynchronous requests are mainly used for write operations. If there is too much writes to the same attribute, the performance of access to the attribute becomes degraded.

The synchronous requests utilize the *boost::future*[10] concept inside *SyncClientRequest* class. The concept is composed from two important parts:

- The *future*
- The *promise*

The future represents reference to value that is going to be defined in future. The promise represents the value itself. The future can be obtained from promise instance. The value of a promise can be obtained via its future *get()* method. If the promise value is known then it is immediately returned. If not then the call to future's *get()* method blocks until the value of associated promise becomes known. The same result can be achieved directly by using mutexes, but the future/promise concept is much more clear and easier to understand.

The first step is to instantiate *promise* for value containing the result of the operation. This is done at the time the request is being scheduled by call to *scheduleSyncRequest* or *scheduleAsyncRequest*. The second step is to instantiate a *future* from the newly created promise. The *scheduleSyncRequest* then schedules the request and then calls *get()* method of the future. This way the blocking API calls to database engine are implemented. Upon completion of a request by call to *acknowledge*, *reject* or *fail* method, the request Result can be one of the following values:

- *WS_RES_SUCCESS*
- *WS_RES_REJECTED*
- *WS_RES_FAILURE*

When processing of a request has been successful. The *Result* member contains the *WS_RES_SUCCESS*. If the operation is supposed to return a value then the value is stored in the *Argument* member. If a request has been rejected by the scheduler, because it was not possible to process it, then the *Result* member contains the *WS_RES_REJECTED* and the request is left untouched. When processing of a request has failed due to internal error. The *Result* member contains the *WS_RES_FAILURE* and the *ErrorMessage* is set to string that represents human readable error representation.

The memory management of scheduler is very important. It can affect seriously whole performance of the database and also can introduce erroneous conditions that can appear later during processing of different requests.

5.3.3 Data storage

The data storage is designed specifically to meet the HLA requirements. Therefore the structure of database records reflects the definition of HLA object classes. Every record is capable to reflect one HLA object class instance and all of its attributes. Every instance of database record can store different amount of attribute values. The count of attribute values that will be stored in the record has to be specified during creation of new database record.

The data storage has to solve two problems:

- Implement record storage mechanisms
- Provide mapping between record key and its stored representation

Solution and implementation of both problems is covered in following sub chapters.

5.3.3.1 Record storage

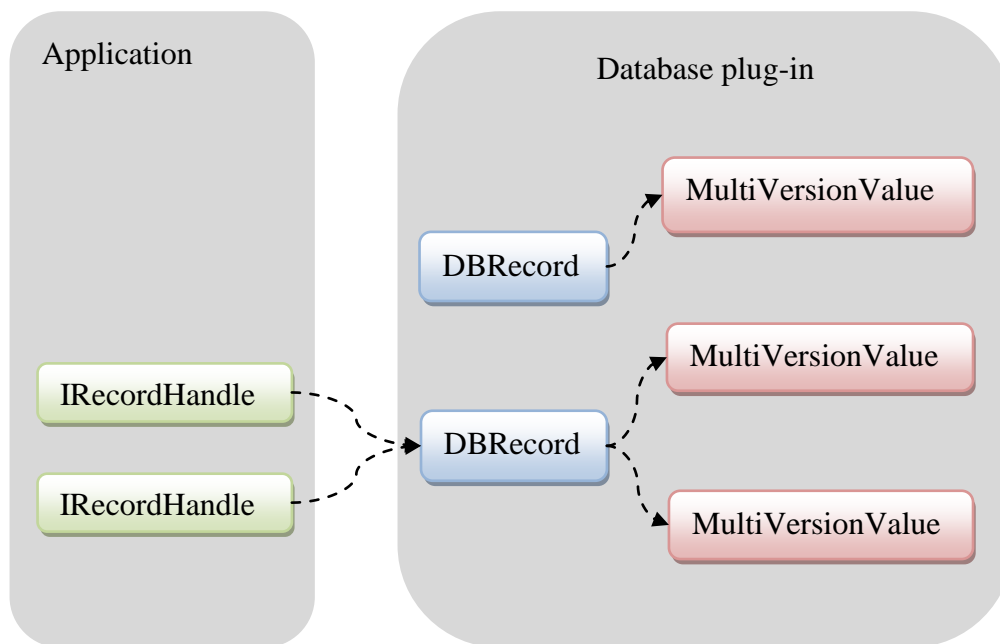


Figure 5-5 Record architecture.

The Figure 5-5 shows relationship between all classes that participates in record management. A database record is represented by the *DBRecord* class. The record class contains member for storing scheduler's metadata, pointers to instances of record attributes. The record class also contains interface for managing the ownership of attributes.

Attributes are instance of the *MultiVersionValue* class, but for simpler explanation it will be referred by simple term the attribute. The storage architecture has been designed with support for database snapshots. For this reason the Attribute class

must be able to store more than one value of an attribute at any point in time. Current implementation of attribute inserts value to the balanced tree and uses timestamp of the write operation as the key. By this way the attribute class can keep all historical values of an attribute. When the value is being read from within a snapshot the attribute class finds the latest known value of attribute that has timestamp lower than or equal to the snapshot timestamp.

Keeping of all historical data is very memory consuming and performance consuming. Searching in historical data can take $O(\log N)$ where N is the total amount of historical values of an attribute. Also the memory consumption is $O(N)$ where N is again the total amount of historical values. For this reason the technique of lazy write has been introduced. The lazy write technique can reduce memory footprint and improve performance in some cases. The detailed description is in chapter 5.3.3.3.

Attributes also provide storage for scheduler's metadata. These metadata are different from the *DBRecord's* metadata, because scheduler needs to know different information when working with attributes. In both cases the data storage does not care about the metadata contents. It just provides space for their storage.

The *IRecordHandle* represents public handle to a record instance. The handle is designed to be passed to an application or database component client. It can provide some public methods (it is a part of Common Interface). The most important thing is that record handles store *boost::smart_ptr[10]* counted pointer to database record. This is a safety mechanism that protects the database record before being deleted too early. The record might be delete through the database interface but as long as there is at least on handle to a record it is not released from memory. An application or a database component client can still schedule requests for the record for as long as it holds its handle.

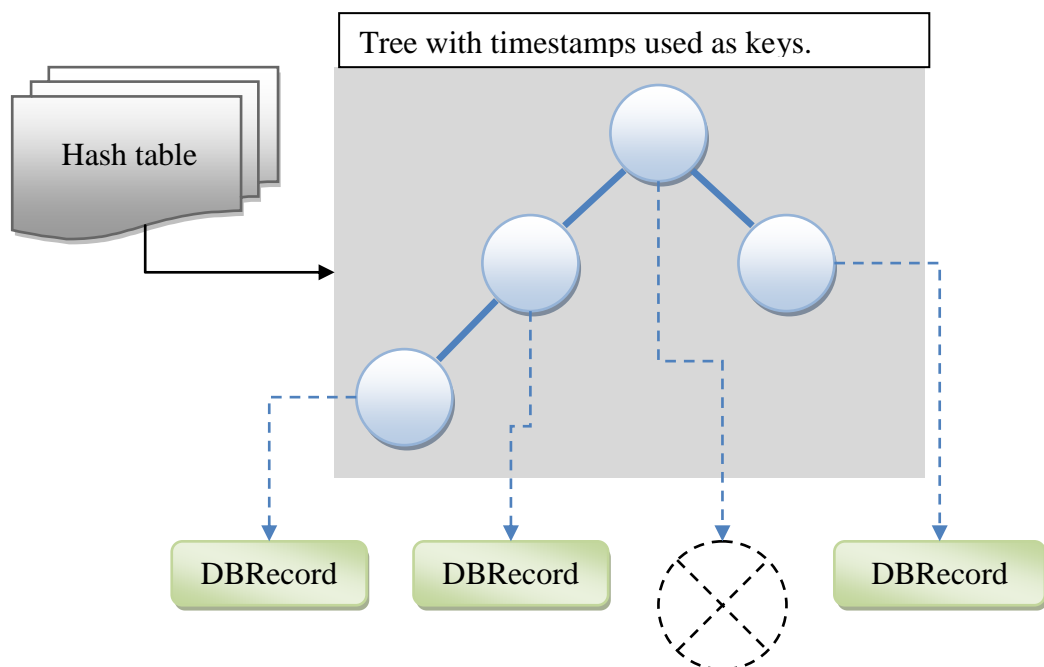


Figure 5-6 Mapping between keys and records

5.3.3.2 Record management

The second important requirement is to manage lifecycle of all database records. It is also necessary to provide mapping between record instance and record key used by the user. The key used for identification of an record is represented by string value.. The reason is again to directly match HLA object class instances design. The key is the HLA object instance name. This design does not need any other translational mechanism for conversion of instance names to database keys. Thanks to the HLA requirements the database keys are guaranteed to be unique.

The Figure 5-6 provides overall view of the key to record instance mapping structures. The first part of mapping is realized by hash table that contains records for every known record key. Hash tables are only reasonable implementation when using keys represented as string. The hash table implementation should achieve approx complexity of $O(1)$. The real performance depends on total number of collisions between keys. The implementation uses default string hashing method, because for proof of concept the performance of key mapping is not the highest priority.

The second part of mapping is designed in the same way as attribute values. The snapshot support requires knowing of old mappings between key and records that were valid at the time of snapshot creation to be able to find historical values. Therefore every entry in hash table maps to a splay tree and. Splay tree is used because it rebalances itself in a way that the most recently accessed nodes are near the root. Therefore during snapshot processing the handle retrieval will optimize the performance of lookups by moving snapshot related nodes up in the tree. The deletion is complicated, because the mapping can't be simply destroyed until the last snapshot that can possibly use the record is destroyed. The deletion is performed by storing NULL mapping as a one of the values. The snapshot support is currently not implemented fully. During implementation of the HLAProxy the priority of snapshot implementation dropped to zero and snapshot has not been required at all (it has been specific feature for continuation project of this thesis). Therefore the splay tree is present, but record mappings store only actual mapping information. The splay tree is represented by `boost::splay_set`[10]. The splay tree uses same technique as the scheduler heaps, it is intrusive container too.

The records are managed by using reference counted pointers, so called smart pointers. The pointers are using implementation provided by boost libraries. The usage of smart pointers is unavoidable because the lifecycle of the database's record is not so straight forward. The record is instantiated during the insert operation and should be destroyed during delete operation. However the record might be still required for some existing snapshot or record handle being held by application. The database cannot destroy such record or the application will most probably crash with memory access violation error. The smart pointers take care of this problem, because the record is destroyed after the last smart pointer is gone. This is important to remember when working with the database source code.

5.3.3.3 Snapshots

The support for database snapshots was the most important feature for project that was supposed to be continuation of this thesis. The use case that was solved by snapshot is long time calculation performed over all entities in the database without blocking of request processing. This feature is above the analysis of middleware's architecture. As this is not related to this thesis, the implementation has been

stopped. However it might become interesting again in future and therefore this chapter provides explanation of main ideas.

The database snapshot is saved state of the whole database at single point in time. The snapshot must store following information:

- values of all attributes currently present in the database
- mapping between keys and records that exists at the time of snapshot creation

Creation of a snapshot is time consuming and performance ineffective. Therefore the snapshot is not represented by a copy of all required information. The implementation of snapshot is distributed across the whole database and therefore it was not possible to remove it when it was not required any more.

The idea of snapshot implementation is based on following requirements:

- Snapshots must consume as little memory as it is possible
- Snapshots must be created as fast as possible

The memory consumption is optimized by introduction of so called lazy write operation on every attribute (explained later). The lazy write technique optimizes memory consumption and performance of the database. The idea is to store only minimal amount of required values to satisfy the needs of all existing snapshots.

For this purpose the database must know all existing snapshots. The snapshots are arranged in the double linked list, so they can be found when it is needed. The *ISnapshotHandle* is pointers directly to the global list therefore snapshot can be accessed in $O(1)$ by its associated handle. Due to the strict ordering of database operations a snapshot is always inserted at the end of the list, again in $O(1)$.

The lazy write operation is defined by following steps:

- Comparison of write operation time stamp and the timestamp of latest known snapshot.
- If the write operation has higher timestamp the latest known snapshot then the value is rewritten
- If the snapshot has higher timestamp than the latest known snapshot timestamp then the value is stored in the tree containing historical values and the latest known value is rewritten.

This is a performance optimization, because when no snapshot is created the write operation stores new value over old known one and operates in $O(1)$. If a snapshot is created and it is required to keep old known value the value is inserted in historical data tree in $O(\log N)$.

This is also a memory consumption optimization that introduces data sharing between snapshots. If there are multiple snapshots created between two writes to an attribute then all snapshots must contain multiple copies of value written by first write operation. The lazy write technique allows sharing of the last value for all those snapshots. The second write operation will store the value that has to be preserved for all snapshots to the tree with historical data so all snapshots can find it.

Memory consumption of a single snapshot is sum of size of timestamp that must be remembered and all copies of data created during lazy write operations that followed. The memory consumption of a snapshot increases as new write operations are changing attribute values. The maximum size of snapshot is sum of all attribute

values in the whole database. This approach is still better because two snapshots create right after each other will share the full copy of database and therefore per-snapshot memory consumption is equal to 50% of the size of the database.

This simple design has significant side effect. The database accumulates old data and at some point in time it must remove them. There is no simple technique that can easily erase all unneeded attributes' historical values. It is required to find all values related to a destroyed snapshot and investigate if there is another snapshot still referring to them. For this purpose the snapshots are arranged in double linked list. It allows retrieval of time stamps of previous and next snapshots in $O(1)$. The check then can use these values to find out whether the next snapshot maps to the same historical value or not.

The snapshot destroying will be time consuming and requires implementation of garbage collector algorithm that can iterate over all attributes in database and removes unneeded historical data. It is possible to group multiple destroyed snapshots together and perform deletion of multiple values.

This is the main idea behind snapshots that may come handy in future if there will be requirement for snapshot support addition.

5.3.4 Worker threads

The scheduler has unified worker thread architecture. It means that every thread is capable of running every operation required by the database. This design allows easier load balancing, and future extensibility through scheduling more operations to the worker pool. Most of today's database schedulers are still single threaded to get rid of excessive synchronization by using multiple thread synchronization primitives. The HLA Proxy's scheduler works the same way. The requirement to keep strict ordering of operations is not easily solved by multiple dispatcher threads.

The performance discussion can be found in the chapter 10. The performance bottleneck is the dispatcher thread. The single-threaded dispatching of requests introduces performance gap. If the operations are going to be fully parallel then the limiting factor is the single-threaded performance of the dispatching thread. Continuation of this thesis should investigate the possibility to perform request dispatching in multiple-threads.

The main idea is that requests are scheduled to worker threads in a thread safe way, so there is no need to perform additional thread synchronization inside the thread's functions. The dispatch thread schedules requests to worker pool on the readers/writers lock principle. It is possible to schedule multiple read operations at the same time, but only single writer is allowed during request processing. This design delivers better performance for read only operations. According to the analysis of the requirements this is a benefit for prototyping tools, which have to perform lot of read operations. Also a prototyping tool can have multiple threads, where every thread simulates single entity. In such case it is possible that the parallel execution is going to benefit from higher read operations throughput.

All read and write requests are executed in worker threads. This is also preparation for future support for persistent storage. The disk I/O is much slower than memory, but the database does not need to actively wait for result of write operation. It can continue in processing of other requests.

5.3.4.1 Error handling

The correct handling of error situations is not easy in environment consisting of multiple threads. The error handling behavior must be well defined to avoid user confusion or reporting of unusable values. There are multiple possibilities on how to solve the erroneous situation.

The biggest problem is that the error can occur in different thread of execution than the thread that has registered a request. The user expects that the failure of his request will be reported by the exception or return value from the thread that has submitted the request (the *registerRequest* method).

The main question is how we deliver the error to the expected place? We can use return values of functions to pass data between threads. We can throw exceptions or implement some kind of messaging.

The scheduler has some kind of exception/messaging combination. Modeling of error values through return values of functions is not so easily maintainable. The exceptions cannot be easily thrown across multiple threads (it can be done but with significant overhead). The result is that every worker thread catches all exceptions. When exception occurs it is packed to the error member of the request that is currently being processed. The request is rejected and returned to the dispatch pool. The dispatch pool can do some self-correction of the error, or unblocks the user and reports the error value.

5.3.4.2 Scheduler's thread synchronization

Almost every multithreaded application has to be synchronized at some points of its execution, otherwise race conditions can occur. The database has to be precisely synchronized, because race condition between internal threads can introduce data inconsistency. Unfortunately the synchronization between threads costs CPU cycles and resources. Thus degrades performance. The HLAProxy uses two types of locks to optimize the performance of the locking and unlocking operations:

- Regular mutexes provided by the operating system
- Synchronization based on atomic operations.

The regular mutexes might introduce degraded performance depending on their implementation by the operating system. On the other hand the thread can be suspended for any arbitrary time without consuming CPU cycles. Some systems use hybrid approach (for example the Solaris OS), when every mutex acts as a spin mutex for some short amount of time and then blocks the thread in the usual way.

The HLAProxy uses *Boost.Thread*[10] library to perform thread related operations. This solution is more portable and does not require any explicit knowledge about the lower level thread API that is being used. Be aware that *boost::mutex*[10] does not necessarily maps to operating system's mutex. The developers of the Boost.Thread library decided which system primitive is the best for their mutex implementation.

The HLA Proxy middleware uses mutexes in two situations:

- It is expected that the thread will sleep for longer time (a client thread waiting for the completion of the database request).

- Synchronization of access to the request queues. The request queuing can either postpone database thread until a request arrives or can postpone request submitter until there is empty slot in the request queue.

The atomic operations are another thread safe way how to modify the data. For convenient usage the Intel's TBB library has been used. The TBB provides easy understandable syntax (see `tbb::atomic`[11]) and also allows atomic operations on non trivial data types (types that does not match the size of CPU registry). The atomic operations are used in HLAProxy mainly in two usage scenarios:

- Updating counters.
- Implementation of locks in situations with low probability of thread collisions.

The statistic counters are good example of counters that does not need to be completely synchronized. The goal is to achieve consistent increment/read/write operations. The ordering of the operations does not matter (the user will get a slightly out of date stats, but that is all right). In such case there is no need to provide mutex that will guard the counters. The locking can be significant performance hit, because parallel threads must wait for each other to just update the counter value. The HLA-Proxy solves this by using atomic operations like "fetch and increment". So there is no thread synchronization needed.

The access to single database record can be optimized in the same way. We can use atomic operations to implement active locking mechanism. Active locking is generally bad idea, because it consumes CPU resources while still trying to obtain the lock. The spin locks are best performing in situation when there is small amount of collisions between threads. The spinlock that guards attribute is prepared for synchronization of garbage collector required for snapshots. It is currently unused.

Atomic operations are also used during execution of database operations during reading or writing to an attribute. Attribute's metadata contain members *ActiveReads* and *ActiveWrites* (for metadata structure definitions see chapter 5.3.2.2). The dispatcher atomically increases correct counters (depends on type of request) before passing a request to worker threads. After request execution is finished, the worker thread atomically decreases correct counter value (depends on type of operation read or write). It doesn't matter how many parallel request have been scheduled, due to the atomicity of counter operations one of the worker threads will decrease counter to 0. The thread that reaches the zero value in a counter is the thread that finished execution of all currently executed parallel requests as last. The thread then obtains queue mutex and wakeups waiting requests (only those that can be processed usually N read requests or single request of any other type).

This model of synchronization guarantees correct transfers of requests between queues. It is also performing better because dispatcher and worker threads lock the queue mutex only when it is required.

5.4 FOM Abstraction Module

The Figure 5-7 represents internal architecture of FOM Abstraction Module. The blue arrows represent data flow between various parts. The figure shows following relationships between components and an application:

- The HLA component has bidirectional interaction with Database component. That is because the HLA needs to store and retrieve attribute values.
- The application has bidirectional relation with HLA component because the HLA component realizes the RTI communication module. The client can register interaction callbacks therefore HLA component can communicate in opposite direction.
- The client has bidirectional relation with Database. The client has to query the database too, for example during interaction processing. Also the client can do some calculations on the background directly through the database API.

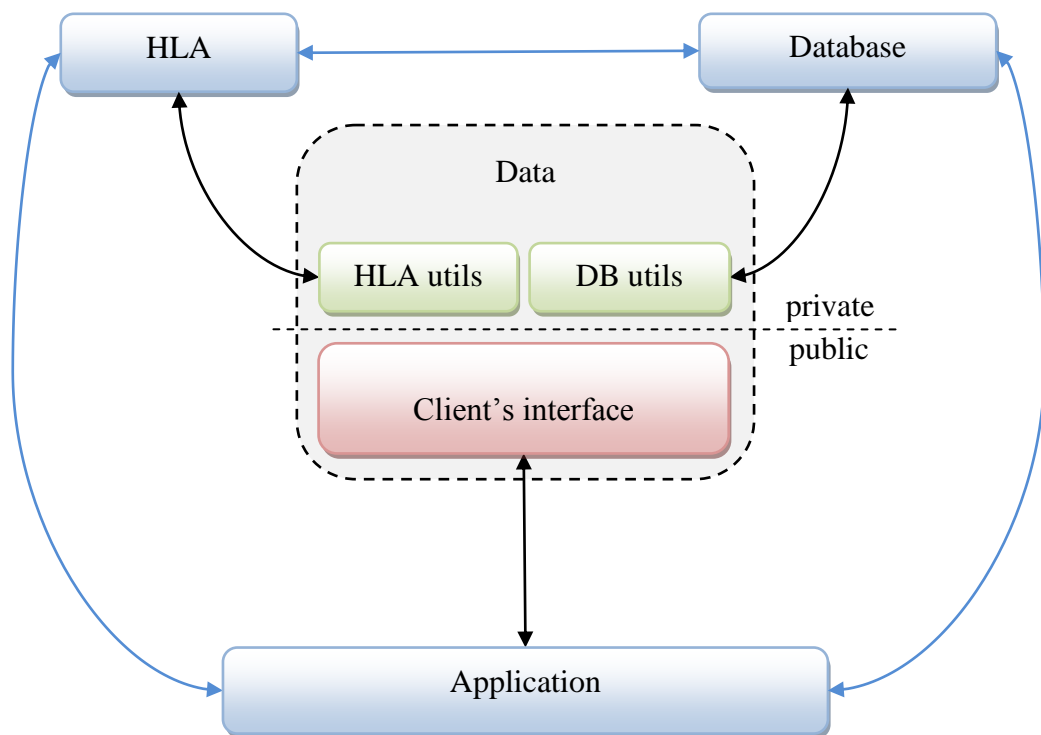


Figure 5-7 FOM abstraction layer implementation

In the middle of the figure, there is data model definition called simply the Data library. It is not a component, because it cannot perform any task on its own. The black arrows mark the real interaction realized by function or method calls. The main goal of Data library is keeping whole data model in single place. The data model has to contain following parts:

- The HLA helper utilities (encoders, decoders, ...)
- The Database helper utilities.
- The data object proxies that define publicly accessible user interface to HLA object classes and interaction classes.

This design allows keeping all parts related to data definition in a single place. The benefit is that we can easily replace the data model at any time. In fact the

Data is implemented as a shared library (but not a plug-in). When the HLA data representation changes there is need to only rebuild one shared library.

When a new component is introduced that needs to access the data, the design of the component shall follow this design principle. Otherwise the new component can easily break the centralized data definition.

To protect internal data model structures against improper usage the two levels of access are declared:

- Private – The private interfaces are available only during compilation of the middleware. Only middleware's components can access it.
- Public – The public interface is accessible to everyone. This successful only interface that is visible to the middleware's client.

The various parts in the Data model communicate directly with each other. Therefore it is required to instantiate and setup components in specific order. Current order in this thesis is following:

1. The user must instantiate and start database
2. The user must instantiate and start HLA
3. The user can use the data interface.

The lower level details are mentioned in separate chapter about implementation of specific parts.

5.5 *Data shared library*

The data shared library contains implementation of the data model discussed in chapter 5.4. This chapter describes data library internals in much more details. The Figure 5-8 is explained in following subsections.

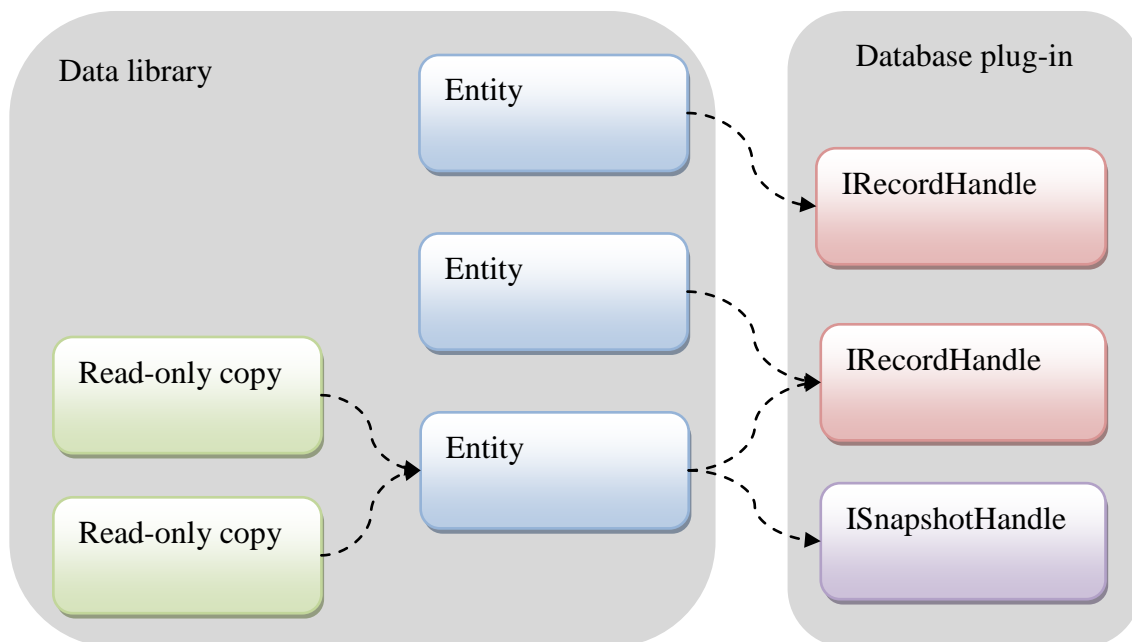


Figure 5-8 Relation between data classes and database records

5.5.1 HLA object instances

The classes, which represent HLA object instances, are designed for providing user friendly interface to the database records. The Figure 5-8 shows simplified view of relation between object instance representation (Entity class) and its associated database record. In reality the Entity class is represented by hierarchy of C++ classes that exactly match the HLA object classes defined in FOM. For illustrational purposes the simple “Entity” name is present, but it can be any class defined in the FOM (e.g. Player or Bot).

Every Entity class is derived from *BaseEntity* class. An Entity class is associated with single database record. The entity cannot exist without an underlying database record and the association between entity and record remains unchanged for the life time of the Entity instance.

The mapping between entities and records is realized only in a single direction. Every entity stores internally *IRecordHandle* of its associated database’s record. When two entities points to the same record it does not mean they are sharing same record handle instance. Each instance has its own private handle to the same database record. This design allows accessing of the record concurrently from multiple threads. The entities can’t share same handle, because the handles participate in database memory management and record locking mechanisms (See chapter 5.3.2.3 for more details).

Every entity can be mapped to single database’s snapshot. The snapshot mapping is not mandatory. When there is no mapping to existing snapshot the entity behaves exactly as mentioned in the paragraph above. If the entity is also mapped to a database snapshot the entity becomes read-only. The database does not support writing to snapshots. The main functionality remains unchanged. Every access to Entity class members results in read operation on specific database snapshot.

The Entity classes serve as a proxy for the database records. Every request to set or get value initiates database request for record operation. The operations on the entity are atomic only when working with single attribute. It is possible to lock an entity for exclusive access by using lock/unlock methods. The Entity class does not need to implement its locking mechanisms. It just locks the underlying database record handle. Only a single entity can hold the lock and only the lock owner can unlock it. Improper usage of locking mechanisms can lead to deadlock of the middleware. If lock is called and the entity is not able to obtain the lock then the lock method blocks its caller until the lock is obtained. When the entity is locked then all operations on other entities would block its caller until the lock is released. The order of processing of waiting operations is based on the database’s scheduler implementation.

The entity is aware of current ownership of HLA attributes. Attempt to write to read-only attribute does nothing. The user has to obtain the ownership of the attribute. The read-only flag is shared between all instances. When an attribute becomes writeable then all existing entity instances, which are capable of writing new values, start writing to the database automatically.

The entities can’t be instantiated directly by the user. This limitation is due to the runtime linking options on Windows operating systems (See for explanation of various linking problems Appendix D). User has to use HLA interface to create entity instance and then release the entity by call to its release method. This is the only supported and safe enough way that works across all supported platforms.

The entity classes are located in files with prefix “obj_” located in the data directory.

5.5.2 The HLA read-only object instances

The read-only entities are marked green on the Figure 5-8. One of the usual usage scenarios of the middleware is to process the entity’s attributes at some point of time. During that time a user needs to have copy of all entity’s attributes, which does not change. This can be achieved by creation of a database snapshot. Such implementation would consume too much memory and performance.

The read-only class is snapshot of single entity at one point in time. The user can create a read-only copy of the entity by invoking the *getReadOnlyCopy* method. The entity then locks the record, fetches all attributes’ values to local copy and unlocks the record. The returned type is different, because read-only copy does have only getters.

Every read-only class is derived from *BaseReadOnlyEntity* and the hierarchy is the same as was mentioned in chapter 5.5.1. The read-only classes are automatically generated. Their names starts with RO_ prefix and are located in files with “obj_ro_” prefix in data directory.

The read-only classes can be copy constructed from other read-only classes. The relation on figure Figure 5-8 is for illustrational purposes only. In fact once the read-only entity is created it is no more related to any other entity or database record. The memory management is limited in the usual way. An application user cannot delete the instance directly but must use the release method.

5.5.3 The HLA interaction classes

The interaction classes are in many ways similar to entity classes. Their hierarchy reflects FOM interaction classes hierarchy. The files are located in the same directory as entity classes are, but with prefix “int_”.

There are two main differences between interaction classes and entity classes:

- memory management rules
- parameters marshalling and unmarshalling

An interaction class exists only at a specific time, during the call to the interaction handler form the HLA interface. If the interaction is received through the HLA then the HLA plug-in is responsible for memory allocation and cleanup. The interaction class instance exists only during the handler’s registered callback execution.

If an application wants to send an interaction, it must instantiate specific interaction class and fill in its parameters. Interaction is send by a call *sendInteraction* method on the HLA interface. After the call the interaction can be released, as usual, by call to release method. The difference between object classes is that interaction can be instantiated as a local variable during call. The reason is that HLA component never writes to an interaction class instance during *sendInteraction* method call.

The parameters encoding and decoding has the main logic directly implemented in the data library component every interaction class instance can decode/encode its own parameter values. The HLA component takes care only of sending or receiving of an HLA interaction.

Every interaction is derived from *BaseInteraction* class.

5.5.4 Value marshalling and unmarshalling

One of the most important functions in data library is value marshal/unmarshal from and to a network format. The HLA is designed to allow transfers of any type of values. This is achieved by using only variable length data as values. The RTI does not care about the actual data type, it is up to the federate developers to perform marshalling and unmarshalling of the values. The IEEE1516 standard defines how should be values encoded for some basic data types. In case of complex types it is up to federate developers to define some common binary representation of their specific values. The IEEE 1516-2010 adds support for so called EncodingHelpers.

Encoding Helpers id RTI's interface that allow encoding and decoding of values from and to Variable Length Data, which are transferred by the RTI. The current interface supports following helpers:

- Basic data types (int, string, char) in both big endian and little endian form
- Fixed array of elements
- Variable length array of elements
- Fixed records

The data library provides its own marshaling and unmarshaling code. But for the basic types it utilizes IEEE 1516-2010 encoding helpers. Another layer ofmarshallers allows developers using the HLAProxy middleware to fully customize how the encoding is being done. It is not required to use the provided marshalers and unmarshalers.

The second functionality in data library is to automatically select correct marshaller and unmarshaller for provided attribute or parameter handle. This part is automatically generated from the AI FOM document. The data library realizes mapping from class and attribute handle to specific marshaller and unmarshaller.

The data library contains functions with names `__marshall_X` and `__unmarshall_X`, there X is the name of AI FOM data type. These methods take the data received from RTI and variable of any type as arguments. The conversion is being done from data to value and vice versa. Functions for basic types are pre-implemented in the library. Functions for complex types are automatically generated during build of the library.

The `__attr_marshall/__atr_unmarshall` are only exported functions for the data library. They take class handle, attribute handle, data and value (as in previous case) and call correct unmarshaller or marshaller. For specific purposes the `__marshal_NULL` can be used to disable automatic conversion for attribute. The same applies for interaction too.

This is heavily used in the HLA component because it can then automatically unmarshal received attribute updates or interactions and then send them to the DB without unnecessary interaction or construction of data classes. The same applies for the value publishing through the RTI. Current implementation is not thread safe, but that is not a problem as all marshalling/unmarshalling is being done directly in the RTIs internal threads.

It is questionable if parallel marshalling or unmarshalling has a big benefit to the middleware's performance. The data has to be written to the database in specific

order so the speed improvement by using parallel unmarshallers might not be so significant.

The automatic generation of marshaling and unmarshaling code is not straightforward as it might seem. Therefore the middleware does not support all possibilities and limits the type definition. There are lot of corner cases that need special treatment. For example, the handle marshaling can be done directly, but unmarshaling requires access to the RTI ambassador. The middleware provides example of how to overcome such situations, thanks to the `boost::bind` construct. However more complex constructions like recursive structures, handles inside structures are not implemented now. The thesis does not need such complex types. Therefore their implementation has been left for future work on this thesis.

5.5.5 Generating source code

The data library requires XSLT 2.0 processor to be able to compile successfully. The XSLT 2.0 support is necessary for outputting multiple files during single transformation. In our case one XSLT transformation can generate whole object class hierarchy.

The XSLT code is not so much special, it simply reflects the FOM object and interaction classes to C++ source code. The most important document is `typere-solv.xsl`. This transformation file is designed to be included to other XSLT documents. Its purpose is to serve as a automatic type resolution library.

The template tags are designed to search through the `dataTypes` section of the FOM XML document and find declaration of a type. Once the definition has been found it is automatically converted to equal C++ type that is compatible with HLA Evolved encoding helpers API. The `typeresolver` works in three modes:

- *ConvertToCPPTType*, which finds the storage type representation.
- *ConvertToCPPArgIn*, which finds the type of input parameters for object instance attribute setters.
- *ConvertToCPPArgOut*, which finds the return type that, is used by object instance attribute getters.

The XSLT document does not offer full support for any arbitrary type, because it was not necessary for this thesis. The style sheet document has to be extended for additional types in future (for example array types).

The second problem during automatic generation of source code is to resolve name collisions. The easy solution is to translate the names from the FOM directly to class names by replacing only some characters. This idea has been proven to be bad. The resulting names are not intuitive or usable during development. The decision has been made to keep the names simply with possibility of collisions. For small projects it is not a problem to design FOM that contains non conflicting names. For large scale world the probability of collision is higher and it may be required to introduce better mechanism of name conversion. To create FOM document compatible with the middleware presented in this thesis always use unique names of types and classes. The class name must be unique from last dot to the end of the string.

Do not include MOM classes and interactions into the FOM document. Inclusion of MOM classes will result in compilation fail, because the MOM is using complex structures like various handles inside classes. The type system support of the middleware is not at such level. The middleware is not a general purpose HLA mid-

middleware and there is no reason to use MOM classes to achieve this thesis's goals. The MOM support is enabled, but the code that accesses MOM classes is hidden inside the middleware's implementation and does not utilize the automatic source code generation.

By default all MOM classes and interactions are merged with the FOM document automatically during startup thanks to the modular FOM support of the MAK RTI. Therefore there is no reason to include them again in the FOM.

5.5.6 HLA private interfaces

The generic code that is same for every deployment of HLA architecture is placed in the HLA plug-in. The private interfaces in data library are designed to solve following situations:

- Automatic subscription of data classes through RTI.
- Transparent un-marshalling of updates that were received through RTI.
- Transparent marshalling of updates that shall be published to the HLA.
- Handle mapping cache.

Automatic subscription of object and instance classes is done according to a very easy algorithm. The XSLT transformation generates subscription for every leaf class in the FOM object tree that has sharing set to Subscribe or Publish/Subscribe. There is no reason to subscribe other classes. The same mechanism is applied for publishing of the classes. If a middleware's client needs to provide specific subscriptions then it must be done through HLA plug-in interface.

The un-marshalling and marshalling of data updates is handled through BaseEntity interface. Every generated class must implement pure virtual methods that must perform unmarshalling or marshalling of the data according to the class attributes types. The HLA plug-in only access the data classes through BaseEntity interfaces, which can be obtained from class factory and handle cache.

The handle cache is the most important part of the data library from the HLA plug-in point of view. The handle cache is basically a set of hash tables, which can perform various mappings. The handle cache is populated when the HLA plug-in is instantiated and calls initHandles. The generated code then obtain handles of all object classes, instances, attributes from the RTI and populates internal cache tables. At this time the mapping between HLA classes and database record attributes is created.

5.5.7 The Database private interfaces

The database interface contains only simple method. The method is used for storing pointer to currently running database instance. This pointer is shared by all instances of classes that need to access the database.

If the database startup is successful then the database pointer is valid and updated through the database private interface.

5.6 *Application specific support module implementation*

The Application specific support consists of various set of components. The set of components evolves over time. Currently the application specific support module consists of following components:

- Configuration management
- Log output

5.6.1 **Component for configuration management**

Every complex project requires some configuration mechanism that allows easy administration. The HLA Proxy configuration module was designed to accomplish following goals:

- The configuration plug-in can be replaced in future for more suitable implementation.
- The storage of configuration data is centralized for all components.
- The storage does not define how the component stores its private configuration data.
- The configuration management does not require complex tools for its administration.

The structure of the configuration data reflects the architecture of plug-ins and components. A component may have its configuration record stored inside the configuration database. The component configuration record consists of pairs of values. Every pair is a mapping of property name to property value.

The configuration can contain only one configuration data per component type. This might seem as a limitation in functionality. However during the implementation of the middleware, no configuration per component instance has been required. The problem is that component instances has no IDs, therefore it is not possible to store their per instance configuration data.. The full support for per-instance configuration will require extending the PlugEngine architecture and also the configuration component. For now this is not considered as a significant limitation in functionality.

The configuration component interface allows obtaining property values from the configuration database. The configuration component treats all property values as string and does not associate any meaning to the value itself. Such solution does not allow validation of configuration data and forces the middleware developers to implement its own validation and parsing of the property values.

The configuration component also does not allow storing new values to the configuration database. The reason is that the HLA Proxy must be configured before use and during the runtime execution it is not possible to alter the middleware's configuration. The configuration database is designed to describe start-up state for the whole middleware. The components are free to change their configuration state locally during execution.

The storage of the configuration is written in the XML language. The XML is heavily used in today's applications and therefore there are plenty of tools for XML manipulation. The configuration (as was mentioned before) is independent on the

underlying data storage. There are multiple options of how the underlying data storage can be implemented:

- Plain text – The main disadvantage is that the configuration component has to implement more functionality and the final solution is equal to simple XML based configuration. On the other hand it is best suitable format for convenient configuration management with simple text editor.
- Database – There are small SQL engine implementations, but usage of such solution was considered as a big overhead compared to the configuration data complexity and usage scenarios.
- XML file – The XML language is somewhere in the middle and therefore was chosen as default data storage protocol. Firstly, it should be possible to create GUI application for XML management or to reuse some existing software. Secondly it is possible to create XML Schema definition and validate the XML file before parsing it. The XML Schema is not supported in this initial implementation.

Every component obtains pointer to the plug-in manager instance during its instantiation. Right after the plug-in manager instance is discovered by the component, it is possible to obtain configuration component instance and retrieve component's ion data from the storage. This is the place where the property values get parsed and validated. If the configuration is invalid, the component can signal failure during its configuration phase and plug-in manager will perform automatic cleanup and report of the failure.

The configuration plug-in is not thread safe and the configuration component is implemented as a singleton. This allows easy configuration sharing but such solution is not thread safe. Because the configuration is read-only it is safe to work with configuration component from multiple threads. However the best scenario is to initialize the whole middleware and all required components from a single thread.

Example below shows configuration file for two components:

```
<?xml version="1.0" ?>
<configuration>
  <interface name="IcontextLog">
    <property name="STD" value="IlogStdout" />
    <property name="FILE1" value="IlogFile:file1.txt" />
    <property name="FILE2" value="IlogFile:file2.txt" />
    <property name="DEBUG" value="IlogStdout" />
  </interface>
  <interface name="IHLADataBase">
    <property name="FederationExecution" value="aifom" />
    <property name="FederateType" value="HLAProxy-federate" />
  </interface>
  <property name="FDD" value="aifom.xml" />
  <property name="log" value="STD" />
  <property name="TimeRegulating" value="true" />
  <property name="TimeConstrained" value="true" />
  <property name="id" value="HLAProxy-TEST" />
</configuration>
```

Example 5-3 Configuration file

The *ILogContextLog* configuration is an example of how can per instance configuration be stored in the configuration database. For the full explanation of how the per-instance configuration is used please see section 5.6.2.

5.6.2 Output logging component

During either usage or development of an application there is always requirement to provide some output. Every user or developer has his requirements on where the output should be stored and how. Because of this the HLA Proxy middleware does not provide only a simple API, but contains component, which is aimed at performing the output of log messages.

The Figure 5-9 shows the log architecture. The main component is called *ContextLog* (its Common Interface has name *ILogContextLog*) and provides high level log output. The messages that shall be logged are separated into different *contexts*. Every context has assigned a *logger*, which realizes the output of messages. The HLA Proxy middleware contains these loggers:

- *Null logger*, which does not provide any output and only consumes the messages. The logger is used by default when there is no other associated logger for the context. Therefore default behavior of the middleware's logger is to discard everything when not properly configured.
- *Standard output logger*, which outputs messages to the standard output. The logger is singleton and therefore is shared between all contexts of the same type.
- *File logger*, which outputs messages to a file. The logger is instantiated for every context that has to be logged to the file. The assignment of the same logger instance to multiple contexts is possible, because the logger is thread safe and outputs single line at a time. It is also possible to create two instances of *File logger* with the same file; this usage is forbidden even that it is possible.
- *Ncurses logger*, which is designed to support logging into a ncurses's window on Linux operating system.
- *Source logger*, which is specific implementation for Source Engine and is discussed in separate chapter.

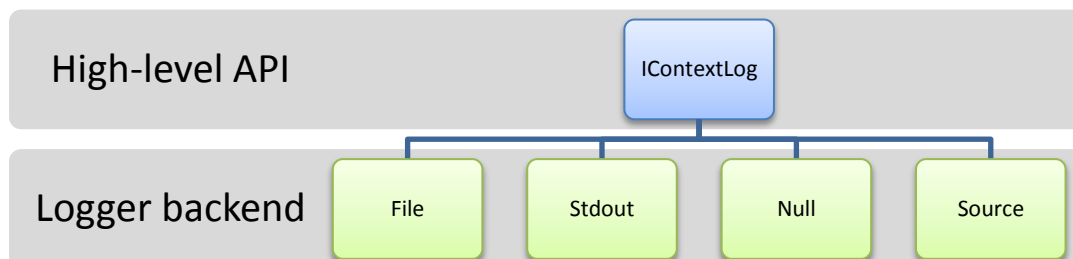


Figure 5-9 Architecture of output logger

The log output component can be extended by creating a new *logger* if it is required. Every logger must conform to the *Ilog* Common Interface definition. If the new *logger* is not inside the log plug-in, the plug-in that contains the logger must be loaded before creation of the Context logger instance (that means before the PluginManager's method `createInterface` is called for the first time). Violation of this rule will result in failure of the Context logger instance initialization.

This design allows configuration of multiple contexts in the HLA Proxy configuration. Every other part of the HLA Proxy accepts only a context name and uses the `IContextLogger` interface to store its output. The result is that multiple components can either share the same context for output, or have specific context for every component. The user of the middleware can easily reconfigure the whole HLA Proxy log output through the middleware's configuration.

The configuration is obtained automatically from the configuration plug-in during the *ContextLog* instance initialization. You can see example configuration file section below:

```
<interface name="IcontextLog">
  <property name="STD" value="IlogStdout" />
  <property name="FILE1" value="IlogFile:file1.txt" />
  <property name="FILE2" value="IlogFile:file2.txt" />
  <property name="DEBUG" value="IlogStdout" />
</interface>
```

Example 5-4 Log output configuration

The configuration section tells the *ContextLog* instance to register four contexts with names: *STD*, *FILE1*, *FILE2*, *DEBUG*. The Context log then parses value attribute to obtain the name of the *logger* component instance. Every logger instance is obtained in the usual way through the *PluginManager* instance. Then the configuration part after the ':' symbol is passed to the new logger instance. This allows passing arguments to the loggers. It is a responsibility of the logger component to parse and understand the argument section of the string.

The whole output logging mechanism might look complex but it is for a good reason. At the beginning of development there was only a simple logging interface not wrapped inside a component. This approach has introduced lot of complications and required rebuilding of the whole middleware when logger code changed. Therefore all log code has been wrapped into components and stored in a single shared plug-in. The problem seemed to be solved, but during work, the problem showed up again. This time it was impossible to distinguish outputs from components and the debugging take a long time. To solve this issue the *ContextLog* mechanism has been introduced.

The last thing to be noted is thread safety of the backend loggers. To obtain reasonable log output from multi-threaded application, the output of the messages has to be serialized. Otherwise the log output will be result of thread interleaving. Some of logger back ends might not need to solve the thread synchronization, because the way they realize message output is thread safe itself. For this reason every backend is designed to log a single line. Every output has to state when new line begins and when line ends. It is up to the logger to ignore these operations or to perform mutex lock or unlock. The impact on middleware developer is following:

1. Always use *ILog::beginl* and *ILog::endl* in your output, when the output has to be done atomically.

2. The *ILog::beginl* and *ILog::endl* has to form a correct bracketing like opening and closing tags in a XML language to avoid deadlocks in backend loggers.

The result is a modular, easily manageable architecture for log output. It can be used by the middleware user too for her application.

6. Available engines

This chapter provides short introduction about few game engines that were considered as one of the possibilities for demonstration of the middleware. Every engine contains some brief introduction and discussion about its capabilities and some personal opinion based on my own experience.

6.1 *Virtual Battle Space 2 (Real Virtuality engine)*

The Virtual Battle Space 2 (VBS2)[14] is a real army simulator used for training of infantry. It is the most powerful solution compared to other game engines. The VBS2 is HLA compatible, because it is required by its military customers. Enabling connection between Pogamut and VBS2 would be very valuable for the KSVI department, because there is a vast amount of possibilities for AI study in the rich VBS2 world.

The biggest complication is modification of the engine. The engine itself is distributed only with some editors and tools for creation of scenarios through its scripting language. The engine can be extended by plug-ins in shared libraries. According to the available materials the engine is designed and ready for more complex modifications.

The Developer edition contains plug-in SDK. However the provided header files do not allow creation of complex plug-ins or access to engine internals. Additional header files are provided separately as a Fusion middleware product, which is quite expensive.

The VBS2 product is protected by hardware key, which is using HASP protection. The fusion middleware replaces main executable and the hardware key protection requires different set of keys to be present. Otherwise it is not possible to launch the VBS2 after Fusion middleware installation.

Working with the VBS2 has shown lot of complications due to the hardware key problems. From time to time keys were failing. Once the fusion middleware keys are written to the hardware key then it is not possible to launch the VBS2 without fusion middleware. The trial key allowed only to execute VBS2 for 100 times, which is not enough for real development and testing.

The engine does not contain direct HLA support. The HLA support is available through extension called LVC Game. The LVC Game is part of the VTK distribution (see following chapter for more details).

As a result the implementation for VBS2 has been postponed due to technical complications.

6.1.1 LVC Game

As was mentioned in previous section the VBS2 is already HLA capable. The question is, it is required to develop a new solution? The answer is not so simple. Technically the HLA interface is there. The VBS2 uses LVC Game[15] interface from the Calytrix Company to enable HLA support. The LVC Game middleware allows synchronization of entities through various protocols (Mainly DIS or HLA). It is a configurable and flexible solution.

In the VBS2 VTK version, which was available during evaluation phase of the thesis, the HLA plug-in for LCV game has been missing. The only supported protocol was DIS, which is considered as obsolete solution by the HLA. Second

problem has been supported FOM model. The LVC Game is designed for military purposes so the default FOM contained entities according to the RPR2 FOM standard used in military simulations.

The RPR FOM standard contains lot of valuable information than can reflect the simulated world. However the VBS2 does not support RPR FOM fully, so there are parts that will be never reflected or published to the HLA Federation. The AI development may require specific information or interactions to be transferred across the HLA federation. This is not possible because LVC Game contains only mapping between internal VBS2 and HLA entities. There is no way how to extend the VBS2 to support specific functionality without modification of the engine itself and most probably the LVC Game too.

Last option is to provide customized plug-in, which will enable VBS2 to use completely customized HLA stack. Both solutions are complicated and can be done at a later time, if the thesis will be successful.

6.2 *Source Engine*

The Source Engine is a product of the Valve Company. The engine was implemented for the Half-Life game series. Today the engine powers modern games like Left 4 Dead and various open-source games. Over the time the engine has undergone a lot of modifications to support new features and platforms.

The engine is still actively updated with new features. Currently it is available in multiple versions: 2006, 2007, 2009, Source Engine MP. The most recent version is powering the open-source version of Team-Fortress 2 game. The 2009 version is powering Half-Life 2: Episode One, Episode Two and Portal game. The two oldest versions are closed. The user cannot create other game content than maps. The 2006/2007 versions are fully open and it is possible to modify the engine itself directly through its C++ SDK.

Over time the Source engine has been rewritten to support multiple threads of execution. That means that the developer can spawn multiple real threads and some parts of the code are processed in parallel (like rendering).

The Source engine looks like a very good and powerful open source option. This opinion has slightly changed during the implementation phase. The main question that should every developer, which considers usage of the Source Engine, answer before adopting the Source engine platform:

- Are you creating a game or advanced modification of the engine?
- Does the Valve's WIKI contain enough of information that you require?

The documentation of free version of engine is still under development so do not expect to learn much about Source SDK internals. The basic mechanisms are described quite well, but detailed information has to be found in the source code by developer himself. The source code of some parts is quite complex and not so well commented. To understand what happens under the hood is quite difficult and time consuming.

The second problem is to identify what part of code does the job you are searching for. For example the older versions of HL2 game utilize node-graph for AI navigation. This has changed in latest releases, and now games utilize the navigation meshes technology. Source engine has support for both of the solutions. In many

situations you will realize that you are investigating the source code of node-graph, which is not used at all. The preferred approach is to design a level by hand, which is again time consuming, then run HL2 in debug mode and inspect everything, before you start digging deeply into the source code.

The biggest problem is server/client code. Every entity has two representations, one for the client and one for the server. This is a common solution and naming conventions are described on the wiki or can be easily spotted. Unfortunately at some places the typedefs are used to share the implementation on both sides. From the first point of view it is hard to distinguish whether the implementation is in client or server and if they are different.

However the biggest problem is entity implementation. Some of the entities can be spawned in client/server environment and some of them not. However there is no documentation containing list of fully working entities so it takes some time to find out, which entities should not be used. Some of the entities can be enabled by implementation of missing parts that enable the entity in client/server environment. Other problems are not caused by entity's implementation, but by its mesh, which is lacking required animations. Such entities can be spawned, but do not move even when the source code is implemented correctly. It is a good idea to not start with Source SDK directly unless you are prepared to do a lot of hard work. Better solution is to choose one of existing mods, because they have addressed many of this issues (for example Garry's Mod).

The thesis implementation is based on Half Life 2: Death Match mod because it is available for free and non working entities are not a problem during this initial stage of implementation. The mod is suitable enough to show the possibilities of the middleware. For production deployment it is reasonable to switch to some more advanced modification of the game or different engine completely. The Source Engine is using statically linked runtime libraries. This shows capability of middleware to operate correctly under very problematic scenario..

My personal opinion is not to use Source Engine directly unless it is needed. The 2007 release is old and complicated to enhance.

The Source Engine is distributed through the Steam. Every user that has purchased any Source Engine based product is eligible to download free Source SDK. The SDK is located in "Tools" section and install is done by few mouse clicks in the tools list.

The Steam platform is also one of the weakest points, because it is not possible to execute your custom modification without being online and logged into the Steam account. From time to time the Steam servers are down and it is not possible to develop your product at all. The only known workaround is to switch account to offline mode before the network outage happens. This workaround helps only partially, because when the Steam's account is in offline mode then the Steam does not allow execution of multi-player games.

The new Mod can be created from the menu that is displayed after launch of the SDK. The Source Engine mod is installed in separate folder and automatically added to the list of local games in the Steam's GUI for convenient launching.

6.3 Unreal Engine

The Unreal Engine from Epic Mega-games is one of the most known game engines. The engine is well known thanks to game titles from Unreal Tournament series, which are known by many gamers.

Today the Unreal Engine 3 powers many game titles and supports many hardware platforms like Xbox, Play Station, iPhone and PC.

The Unreal Engine Development Kit (UDK) has been released freely to the public in 2009 for non-commercial purposes only. The UDK comes bundled with complete set of tools that allows world editing, animation and scripting via the Unreal Script language. The tools are on a more advanced level compared to tools that are available with Source Engine SDK 2007.

The Unreal Engine 3 is a very powerful platform with its own scripting language, which is claimed to be only about 10% slower than native C implementation. Most of the game coding is being done in the scripting language. The scripting language can be extended with native calls to the C++ source code (by binding shared libraries).

The development for the engine can be done either by the Unreal Script or through C/C++ API. Unfortunately the C/C++ API is not free and developers have to pay for a commercial license to have access to the C++ SDK.

6.4 Cry Engine 3

The CryEngine is product of the Crytek Company. The Engine powers one of the world's most successful game titles – Crysis and Crysis 2. The CryTek engine is modern and up-to-date engine designed for today's systems. The engine is comparable with Unreal 3, but it provides not only tools but C++ SDK for free too.

The CryEngine contains various specific features, like real-time streaming I/O, which allows real-time processing of large worlds on a machine with limited amount of memory. The engine is multithreaded and designed to scale well on multi-core CPUs. The engine is also compatible with all today's gaming platforms (Xbox 360 and PS3). One of very specific features is editor that is capable to render the scene in parallel on all platforms. The level designer works on single PC, but can see the visualization on the connected Xbox and PS3 in real-time.

The engine can be obtained for free for non-commercial usage from the Cry-Tek website. The installation is very simple, just extraction of downloaded zip file. The engine requires user account, otherwise it can't be started. The account for free usage can be obtained via developer web site. The process of registration is very simple. To execute the engine a user must sign on with valid account. It is not possible to develop and debug during network outage.

The officially supported compiler is Microsoft Visual Studio 2008 SP1 but the engine contains Visual Studio's Solution for both 2008 and 2010 versions. The default configuration uses custom STL implementation known as STLport instead of standard STL library. The STLport delivers higher performance.

The CryEngine 3 programming documentation is very limited. Lot of information is available through public forums, but it is not easy to select good keywords. The best practice is reverse engineering of the source code like in the Source Engine source code.

The source code seems to be much better compared to the Source Engine source code. Unfortunately reverse engineering of such complex code is very time consuming task.

7. Integration to the Source Engine

The source engine can be extended in two ways. The first is to extend its server with server plug-ins. However this is not so powerful solution. The second option is to develop customized modification of some Source Engine based game (Mod). The benefit of Mod development is a greater possibility to perform more complex source code modifications.

The thesis contains implementation of the HLA Proxy Mod, which is customized version of *Half-Life 2: Death Match* (HL2:DM) is standard multiplayer game. It is not one of the most favorites of today's players but it is good enough to show how the HLA Proxy integration works. The HLA Proxy Mod benefits from HL2:DM because it is a standard network multiplayer game. This allows testing of initial implementation against real players. On the other hand it does not deny implementation of in-game bots. The second reason is that the Source Engine mod can prove that the middleware is capable of running with mixed linking of runtime libraries. The Source Engine has been investigated for longer time than the *CryEngine 3*. The Source Engine Mod contains also custom implementation of AI being that can be in a very limited way controlled from outside world. The implementation shows that it is possible to easily publish any kind of data and for the proof of concept is sufficient. Unfortunately for the real deployment of the solution much more work has to be done to develop fully HLA enabled non playable character.

7.1 Integration of HLA Proxy middleware

The Source Engine works always in client/server mode even for local player. The integration can be done into the client source-code or server-side source code. There is no benefit of creating HLA enabled HL2:DM client (but it is possible). The enabling of HLA support in server-side code allows full control of current level, to which a player is connected.

The server core implementation is located inside *gameinterface.h* file. This is ideal place to integrate the HLA Proxy middleware in. For successful integration of the middleware to the HL2:DM server, following steps have been performed:

- One time initialization and destruction of the middleware interfaces
- Create synchronization point, so the HLA federates knows that the HL2 instance is in operational state.
- Modify server's per frame handler, so we can postpone the engine or execute customized HLA operations in the main frame updating thread.

Every Source engine mod is composed from four parts:

1. The closed source engine from Valve company
2. The game client located in *client.dll*
3. The game server located in *server.dll*
4. Additional data files

The closed source engine communicates with shared libraries through predefined interfaces, which are defined as C++ abstract classes. The most important class

is *IServerGameDLL* and resides in *gameinterface.h* in the server's source tree. The *server.dll* must provide implementation of this interface.

The one-time initialization/destruction can be easily performed by modification of *DLLInit/DLLShutdown* methods. These methods are called early after the engine is started. Therefore it is not possible to perform here any complex tasks, except of initialization of HLAProxy interfaces and libraries.

The most important thing is to not forget to subscribe/publish correct *Effect* classes. Failing in doing so will result ignoring of data updates, retrieval or NULL pointer returned from all *createInstance* calls.

7.2 Synchronization

The key point of the HLA is time synchronization. Both HL2 and the HLA client must run in time regulating and time constrained mode. This is only way how the RTI can guarantee correct ordering of data updates.

As was mentioned before the HL2:DM runs always in client/server mode. The developer must realize that there are 2 synchronizations ongoing. The player onscreen window is HL2:DM client that is connected and synchronized with HL2:DM server. The HLA enabled server is then synchronized again with the HLA-client on another machine. It is possible that values obtained from local client data might be a bit different from the values seen through HLA. It all depends whether the query is executed locally in client or asks sever.

The HL2 server has multiple points where the synchronization through HLA can be executed. The server differentiates ticks and frames. Every frame is composed of multiple ticks. Every tick the server calls *GameFrame* method. This is not a good point to introduce HLA synchronization, because the *GameFrame* method is very sensitive about its performance. The second option is to hook the HLA synchronization inside frame handlers.

There are two possibilities:

- *PreClientUpdate*
- *Think*

The Server's think is being called at the end of each frame. The *PreClientUpdate* method is called during the last tick of a frame. After some experimentation the *PreClientUpdate* method has been selected as a point for introducing HLA time synchronization.

The per-frame synchronization gives better results, because the engine has multiple ticks for itself and therefore the game performance is not so much degraded. Unfortunately the frame handler is very sensitive to its performance too. If the HLA synchronization takes too long (for example due to processing of many requests) then the performance can become even more degraded (even the application can become unresponsive to player's input).

During some performance testing the reasonable value for RTI's *evokeMultipleCallbacks* has been selected as interval from 0.001 to 0.5. This allows the RTI to simply check its queues and return immediately when the RTI's internal queues are empty without blocking. The upper bound is not important for us. If there are some messages waiting in the RTI's queues then they have to be processed and the engine cannot continue until the queues are empty.

It is not a good idea to publish updates of entity attributes at this time.. It is better to do the message dispatching only by calling *evokeMultipleCallbacks*. This is

one of the benefits of HLAProxy middleware. Thanks to its database with support for concurrent access the entities can update its HLA attributes at any time they wish and then schedule the entity to publish its attributes. This can be done asynchronously by using entity's Think function that periodically sends updates through .HLA. Entities can have different periods between their Think calls. As a result some of them will get updated with higher frequency and some of them with lower. This can improve the performance of the modified HL2:DM game.

7.3 Output logging

The Source engine log plug-in shows more advanced usage of the middleware and provides reference implementation of so called loopback technique. From time to time the middleware has to operate in a non standard environment. For example, it may not be possible to create log files, because the environment does not allow access to the file system. The middleware has to use non standard APIs to perform its tasks. Another example is the need to perform some action in the environment. In such case the middleware has to access data structures/methods of its host.

This chapter shows the loopback technique on the real example. The middleware supports very tunable output logger. However if you run the HL2, you run usually in a full screen mode and also the standard output is discarded. The solution is to implement customized logger that can work with the HL2 GUI. The figure Figure 7-1 shows the loopback technique and data flow during log output.

Every time the HL2 or the middleware itself decides to log event the *Context-*

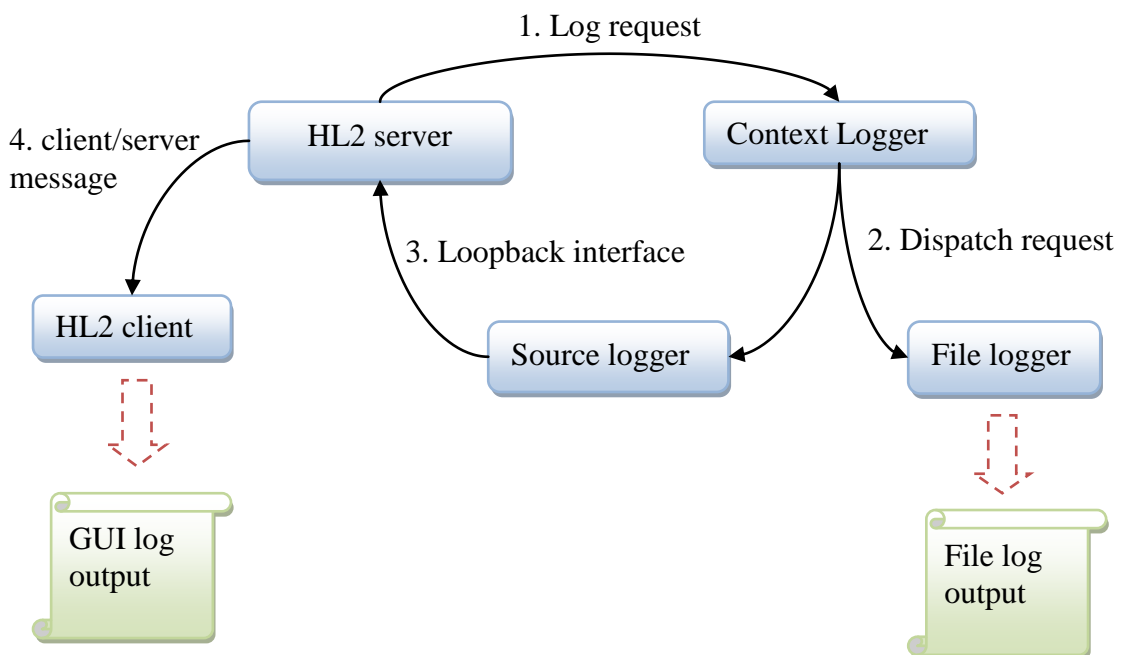


Figure 7-1 Logging via loopback interface

tLog component is used. The component decides to which logger a log message should go (according to the message context and component configuration). The middleware can output data to any logger that can be used in current environment.

The Source engine specific logger is connected to the HL2 loopback interface. Every logged message is passed back to the HL2 server process and then sent to all clients through Valve's client/server protocol. The client outputs the message to the in-game GUI, which is part of the HLAProxyMod.

This implementation shows generic technique how to enhance middleware without modification of its existing components. In this case the message logging subsystem is not aware of existence of HL2. It simply uses backend logger according to the configuration file. However the logger provides more advanced message output.

The limitation of the GUI logger is that every client logs the same messages. This is result of client/server architecture of the Source Engine. Even in a single player game the engine starts internal server and then joins player's client. As the HLA Proxy middleware is integrated to the server and server has no GUI at all, it is required to pass message down to the client. Unfortunately the server does not know to which client the log message should be sent. Therefore the message is broadcasted to all clients and displayed in their logger GUI instance. The logger windows is toggled off by default and can be enabled via console command.

The VGUI definition of the logger GUI is located in `hlproxymod/resource/ui/hlaproxypanel.res`. The gui is very simple and is composed of panel that contains rich text box inside. The implementation is located inside the client source code in `hlproxypanel.h/cpp` files.

The toggle on/off functionality is implemented by introduction of new console variable `cl_showhlapanel` of boolean type. During every `onTick` call the panel checks the value of the variable and reflects the visibility settings. For convenient usage the new console command is introduced with name `HLAProxy_showLog` that takes no arguments. It simply negates the value of the `cl_showhlapanel` property.

The log message is registered in `hl2_usermessages.cpp` by call to `usermessages->register` method and the hooks and associated methods are directly inside the log panel class. The loopback is called `ISourceInterface` and in its `log` method creates custom log message containing string of the message received as parameter and broadcasts the message from the server to all clients.

7.4 Remote controlled NPC

Implementation of custom Non Playable Character (NPC) is a very hard task. The Valve's developer wikipedia contains only a short introduction to NPC development. The implementation of simple NPC has been by reverse engineering most of other available NPCs in the source code. It is not possible to give reasonable description of how exactly NPC development should be done for Source Engine. The resulting implementation has been written without documentation and on trial and error basis.

The HLA Proxy NPC is simple robot that can follow some object. The main goal of for the HLA Proxy NPC has been fulfilled, the NPC can be controlled through console command and therefore through HLA, because the HLA-client (see chapter 9.3) is able to remotely control the engines console.

For detailed information about custom NPC please refer to source code of the HLAProxyMod.

8. Integration to the CryEngine 3

The CryEngine is distributed as a self contained zip file. The SDK is installed by simple extraction of the archive contents. The Visual Studio solutions are located in following directory:

```
<root>\Code\Solutions
```

The solution has different architecture than the Source Engine has. The organization is much cleaner. Before it is possible to build the engine with HLAProxy middleware, the local STLport must be disabled. The local STL provides better performance, but it is not compatible with the standard STL library. The usage of both STL libraries at the same time is highly discouraged as it can lead to several issues. It is possible to combine two STL libraries in single project, but it requires good knowledge of libraries internals and non-conflicting design of both parts of software.

There are only two solutions for STL problems:

- Disable the STLport library in the CryEngine 3
- Rebuild the middleware with STL support.

The middleware depends on MAK RTI library so it might not be even possible to use different STL in the middleware. The safest option is not to use the STLport library. The building the middleware with STLport is left for future work and investigation as it is not important for this thesis.

The switch to standard STL library reduces performance and might break compatibility with other platforms like Xbox, PS3, Linux. Lot of additional work and evaluation is needed for stable release of HLA enabled CryEngine 3.

The CryEngine is built with RTTI disabled.. The result is that `dynamic_cast` is not working properly. The `dynamic_cast` is the only safe way of how to check that the middleware created correct instance of component. The sample implementation did not enable RTTI in CryEngine and is using `static_cast`. The RTTI can be enabled when it is required to use `dynamic_casting` on pointers returned by the middleware.

The CryEngine 3 support is very experimental. The integration has been done only as a proof of concept of the thesis results. Therefore it is not possible to use the suggested 64bit build, yet. The middleware builds fine on 64bit Linux, but it has not been tested to be built on Windows operating systems. For production use it would be best to prepare 64bit built of the middleware and CryEngine 3. The 64bit engine version will allow utilizing more memory.

8.1 *Integration of HLA Proxy middleware*

Integration of the middleware is almost identical to the Source Engine. The first steps are to hook the startup/shutdown functions. The CryEngine 3 realizes these steps in following locations:

- `CgameStartup::Init`
- `CgameStartup::Shutdown`

The source code is identical to the Source Engine startup. The HLA time advanced method is being performed in two places:

- Cgame::update
- Cgame::OnPostUpdate

The second method has been used to perform HLA time advancing. The HLA is ticking every frame. This is different when compared to the Source Engine. The reason is that CryEngine is much more optimized for multithreaded usage and the HLA synchronization is not so much visible. The movement with so high tick rate is still smooth.

8.2 *The Player entity*

The CryEngine is capable of publishing of simple Player class. The integration shows how simple it is to enable entity with HLA capabilities in CryEngine 3. The first step is to extend the game CPlayer class to support the HLA. The following code demonstrates how should be entity initialized.

```
CPlayer
{
    ... usual members
    HLAProxy::Data::Player *pHLAPlayer;
}

CPlayer::setHealth (...) {
    ... usual code
    // --- HLAProxy
    if (pHLAPlayer != NULL)
    {
        pHLAPlayer->setHealth(health);
        pHLA->updateInstance(pHLAPlayer);
    }
}
```

Example 8-1 The example of Cplayer class extension.

Additional attributes can be added to HLA Player class definition and then mapped directly to inside other CPlayer class calls.

This idea can be applied in general to any of the engines entities. Not all entities are being spawned in the same way. Some of them are pre-allocated in memory pools. It may not be possible to rely on entity's class constructors or destructors. The entity might get allocated only once and then continuously reused. Further investigation of CryENGINE 3 is required for detailed understanding of how it internally works.

9. Real deployment scenario

This chapter discusses real deployment scenario that has been implemented to show the middleware in action. The Figure 9-1 shows all components that participate in the demonstration.

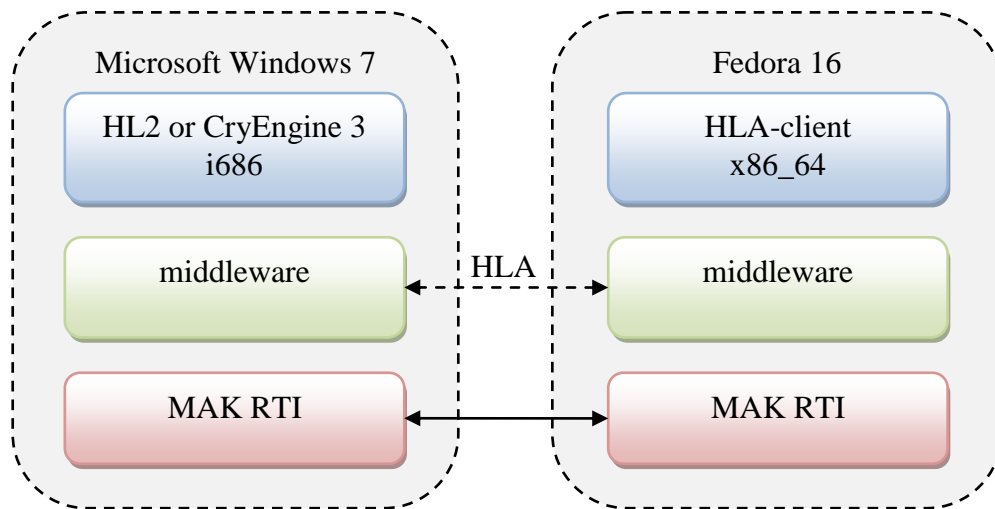


Figure 9-1 Real case deployment scenario

This scenario shows the most complex situation in HLA deployment. Two systems are interconnected through LAN, so the RTI cannot utilize shared memory queues to optimize the network communication. Also both of the systems use different architecture. The Windows platform is running 32bit build of the middleware and the Linux is running 64bit build of the middleware. This selection is intentional to show that the middleware is capable of running under various conditions.

Due to the limited resources the HLA-client has been run inside Virtual Box. One physical machine had to handle everything: the HL2 game, the HL2 client, Virtual Box, RTI and network communication

The simple startup synchronization is implemented and federates must be started in following order:

1. The HLA-client
2. One of available engines.

The HLA-client is configured as a synchronization client and the selected engine as a synchronization master. Improper configuration or order of startup can result in deadlock of federates or undefined behavior.

9.1 HL2 federate

The HL2 federate provides simple data outputs to show the functionality of the solution presented in this thesis. The Table 9-1 shows specific interactions supported by the federate.

| Interaction | Type | Attributes | Description |
|---------------------------|-----------|------------|--|
| ConsoleCommand | Subscribe | Command | <i>A string representing a command that will be executed on HL2 console.</i> |
| MapChange | Publish | MapName | <i>A string representing name of the currently loaded map in the engine.</i> |
| ClientStateChanged | Publish | ClientName | <i>A string representing client's name</i> |
| | | Address | <i>A string representing client's address</i> |
| | | State | <i>State enumeration</i> |

Table 9-1 Interactions supported by the HL2 federate

The HL2 federate is capable of creating and updating HLA object class instances of following HLA classes (FOM document can be find on the attached DVD):

- HLAobjectRoot.SimulationPlatform.Entity.Bot
- HLAobjectRoot.SimulationPlatform.Entity.Player

9.2 CryEngine 3 federate

The CryEngine 3 federate provides almost identical support for HLA as the HL2. The integration to the CryEngine has been done in a short time to demonstrate that the middleware allows easy integration to a third party product. The CryEngine federate has no specific output logging plugin and all logs must be send to a file. Also no AI development have been done therefore the federate only publishes Player object instances and can receive ConsoleCommand interaction in the same way as HL2 client.

The overall performance of HLA enabled CryEngine 3 is far better than the modification of Source Engine. It is most probably caused by better multi-threaded design of the whole engine. The engine runs smoothly without any notice of per-frame time synchronization with HLA-client federate.

9.3 HLA-client federate

The HLA-client is designed to communicate with HLA federates through the HLA. It is a very simple application that shows the idea of remote interaction with HLA enabled engine. The client is designed with simple command line user interface that is composed of following parts:

- Middleware output window
- Standard output window
- Command line interface

The client uses ncurses library to draw multiple windows on terminal. Therefore it is necessary to redirect stdout and stderr output. The simple redirection to a ncurses' window is implemented for *std::cout* and *std::cerr*, so the screen does not get garbled.

The HLA-client also provides simple backend logger implementation that allows outputting of the string to a ncurses's window. Also this is an example of easy extensibility of the middleware. The client application contains another window for *ncurses* logger only. This design allows differentiating between middleware's internal outputs.

The last part is the command line. The client is capable of handling few simple commands. The list of commands and its syntax is presented in the Table 9-2. The *player* and *bot* commands take instance name as an argument. Every command that is not known by the federate is being sent through the RTI as a *ConsoleCommand* interaction.

| Commnad | Description |
|---------------|---|
| exit | <i>Disconnect from HLA and terminates the client.</i> |
| Player | <i>Display known information about specific player.</i> |
| Bot | <i>Display known information about specific bot.</i> |

Table 9-2 HLA-client supported commands

10. Performance

Performance measuring is a very hard and complex task. The biggest problem of performance testing is how to provide objective results that can be achieved in most cases of usual deployment of a software product. In complex systems it is usually almost impossible to provide measurements on all possible setups that users can have. Usual scenario is to deploy application first and then tweak its performance. The same applies to the HLA Proxy middleware.

Another question is: “What is the performance of a middleware?” Well that is a hard question. The performance can be measured as time that has passed between call of a method of the HLA Proxy middleware’s interface and returning of the call back to its caller. There are three scenarios that can happen in this case:

- Call is handled locally in the HLA Proxy middleware.
- Call initiates interaction between federates and must wait for result.
- Call results in invocation of application’s callback.

First type of requests depends mainly on the performance of a database, because many calls perform some data storing or retrieval. Second type of request depends on the whole federation configuration. For example, during time synchronization the call is blocked and waits for the approval from a RTI. The last type of calls can result in invocation of a user provided callback method.

We do not want to measure performance of an application or a RTI. Therefore the only thing that should be measured is the performance of the database. The performance of middleware is affected by other factors. The following list contains items that have the most significant impact on the whole performance of the HLA Proxy middleware:

- The performance of various middleware’s internal components
- The performance of application that uses the middleware
- The performance of selected RTI
- The operating system performance
- The selection of hardware configuration

The middleware is composed from multiple components. Usage of additional components that are not being measured should be limited as much as possible. For example, the file logger can serialize execution of multithreaded tests, because it is using internal mutex.

The performance of the application is unknown and depends on specific deployment of the HLA Proxy middleware. If the application is capable of only a few data transfers the HLA Proxy middleware will work well compared to integration to an application that consumes lot of system resources and sends thousands of interactions or attribute updates. The performance must be measured in a very simple application so the performance really describes the database component.

The RTIs have problems with their performance and therefore every vendor tries to implement the best possible solution. The MAK Company designed RTI that solves the issues by virtualization of the whole network. This means you can put computers in a role of HLA routers/switches to optimize the data transfers. Switching of the RTI to different vendor can have large impact on performance of your federation configuration. The testing application must be connected to HLA during perfor-

mance measurement. However it is not required to call ticks and therefore the RTI will not trigger any callbacks. This is the only way how the measuring can be done without being affected by RTI work.

The operating system and hardware selection is something than could not be configured during measurement due to limited resources.

10.1 Measurements

All performance measurements have been collected on a machine with hardware configuration that is described in Table 10-1. The machine has been running HLA-client on Fedora 16 in Virtual box machine with dedicated 1 CPU. Every test has been run multiple times with various counts of requests. The results contain comparison of single threaded and multi threaded performance. The specific results and comparisons are explained in following specific subchapters.

| Component | Configuration |
|------------------|--|
| Hardware | Core 2 Quad 2.4Ghz (E6600) 6GB RAM |
| Operating System | Microsoft Windows 7 64bit |
| Build target | 32 bit |
| Application | Dedicated application designed only for performance testing. |

Table 10-1 Hardware configuration

Performance measurement has been done for few scenarios, which describe extreme conditions during database access. One extra measurement has been done by using random selection of target attribute and random selection for read/write operation. The random testing can result in collision by writing to the same attribute from multiple threads. This is the best approximation of real usage scenario.

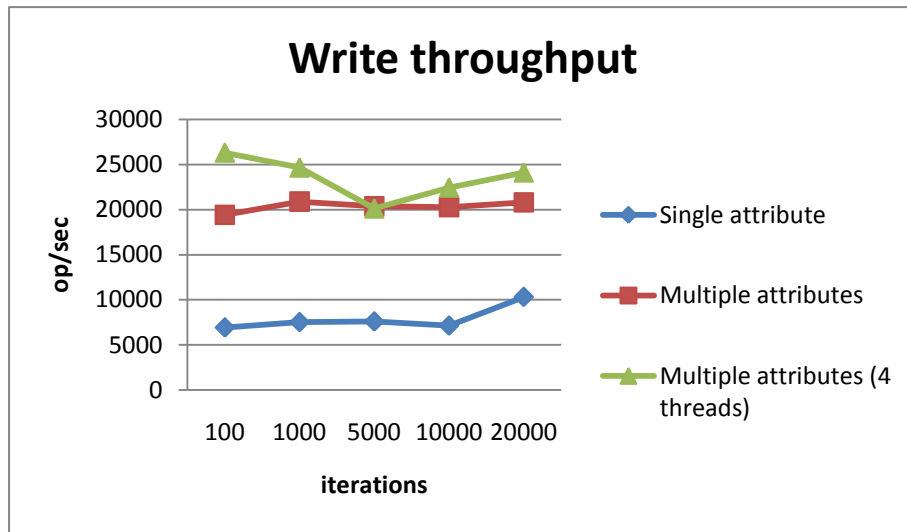
The list of measured corner cases:

- Long series of writes without reads.
- Series of write immediately followed by read
- Long series of reads without writes

Every test contains a cycle that schedules selected operations (in case of multiple attributes it schedules 4 operations writing to these attributes). The iteration count is represented by x axis. Resulting time of execution is then normalized to count of operations processed per second. This representation can easily show whether such performance meets expectations for real-time usage or not. Measurements contain not only time consumed by processing of requests but also time spent in by scheduling of request in application code.

The performance has been measured on release build of the middleware.

10.1.1 Write throughput of the database

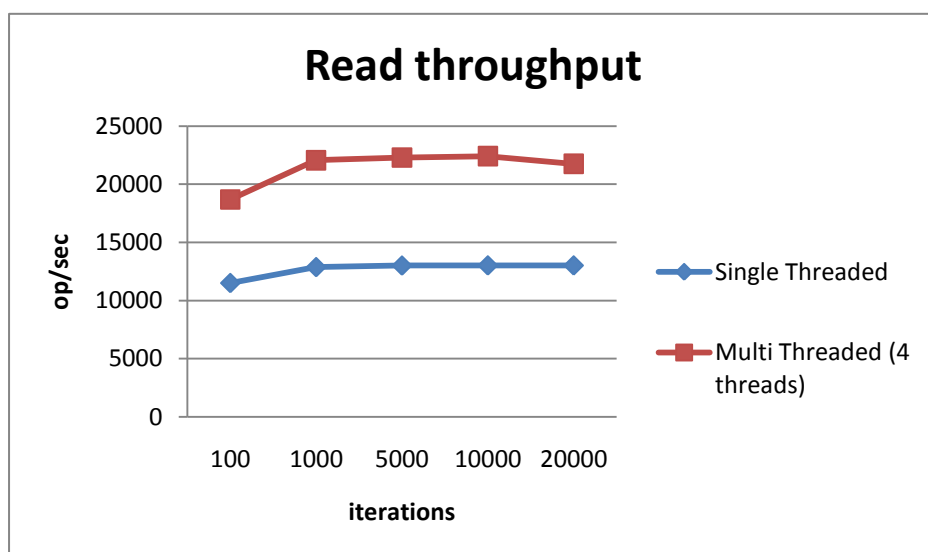


Graph 10-1 Database write throughput

The Graph 10-1 shows performance comparison between various write operations to object class attribute. The write operations are asynchronous and put heavy load on the database's scheduler. The results show clearly that writing to multiple attributes from within single thread has much more better performance. The writing from within multiple threads has slightly better performance than single threaded write to multiple attributes.

The big increase in single-threaded performance is caused by execution of write operations in parallel. The multi-threaded performance is better only slightly. The most probable explanation is that the single thread cannot schedule write requests fast enough as multiple threads, because the usage of database worker threads is the same for both cases.

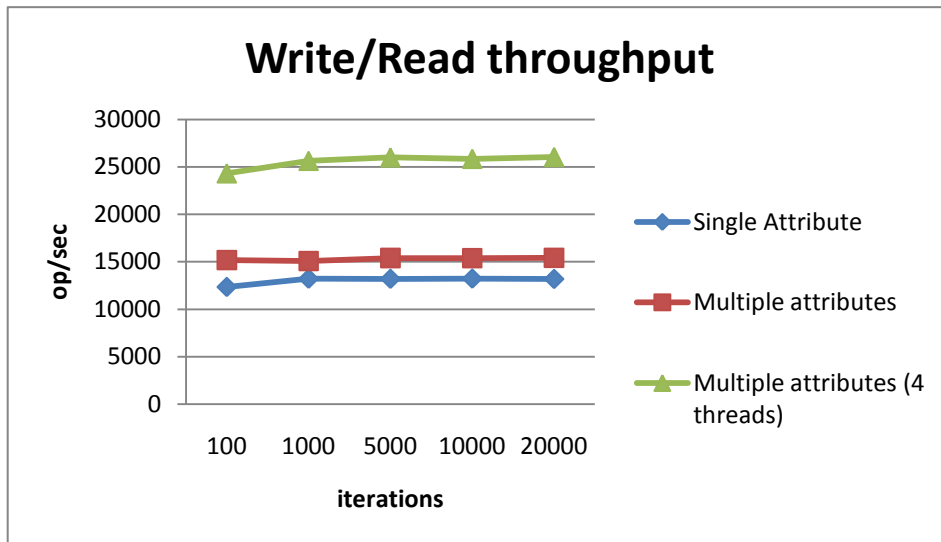
10.1.2 Multiple reads without writes



Graph 10-2 Database read throughput

Read operations are blocking and there is no benefit in reading from multiple attributes from within single thread. Therefore single threaded test for reading multiple values has been omitted in this case. The graph clearly shows that overall multi-threaded performance is almost double when compared to single threaded performance.

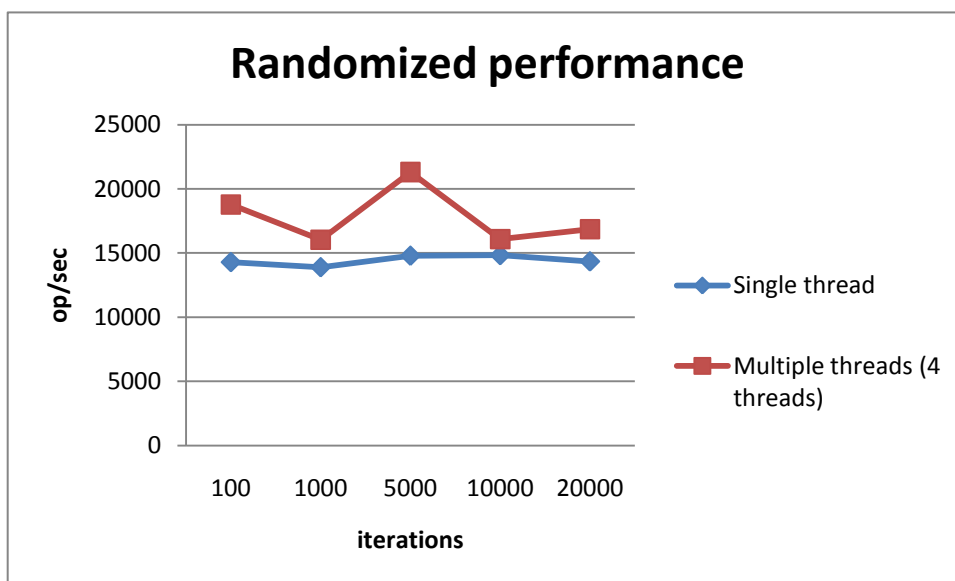
10.1.3 Mixed load



Graph 10-3 Write/Read throughput

The Graph 10-3 shows again that multi-threaded performance is almost double compared to single-threaded one. Operations performed on multiple attributes are slightly faster, probably by faster execution of write operations.

10.1.4 Randomized operation throughput



Graph 10-4 Randomized operation throughput

The randomized test shows that multi-threaded and single-threaded performance is close to each other. This is an expected result, because randomization of operation types and targets has impact on both tests.

In this case the randomization caused scheduling of more writes to different attributes. Such request distribution increases performance of single-threaded operation, because the writes are processed in parallel.

The multi-threaded performance has decreased. Multiple threads can target same attribute and spent the time in blocking reads waiting for each other. This is expected result compared to the corner case scenarios. We can still see that the multi-threaded performance is always better than single threaded one.

10.1.5 Huge amount of attributes

The question is how good the performance is when operations are targeted to more than 4 attributes. This case has not been measured as there is problem in scheduling so many requests.

Theoretically the scheduler will keep only short delayed queues for each attribute and it will spend most of the time with sorting unblocked operations to its delayed heap. This kind of sorting is not optimized as per-attribute queues with hinted insertion. The theoretical result is that the performance will significantly decrease. Unfortunately the database has only a limited set of worker threads and it is not possible to overload dispatcher thread in this way, because only a small number of requests can be executed in parallel.

10.1.6 Conclusion

Performance results collected by simple benchmark look promising. However the maximum possible performance is limited. The performance bottleneck is the scheduler's dispatcher thread. It does not matter how many cores the machine has. The most important information is how fast it is capable of single threaded processing. The significant performance improvement might be achieved by using multiple dispatching threads. Unfortunately such modification will introduce more complex scheduling algorithm and thread synchronization. The resulting performance of database with multiple dispatch threads may also decrease.

Performance of the database will be degraded when the middleware is running inside game engine. That's expected result because the engine is consuming lot of resources. However the middle value of performance equal to 16860.6 (multi-threaded) and 14285.7 (single-threaded) random operations per second is not bad at all. Most engines are limiting its frame rate to 50-60 frames per second. Therefore the database can perform around 250 operations per single frame. This should be enough to reflect selected entities fast enough. If the performance becomes degraded the updates can be sent at lower rate.

The overall result is that the middleware, which is meant to be a prototype only, is reasonably powerful according to some simple scenarios measurements. The performance meets expectations for usage of the middleware for interconnection between agent based development platform and prototyping tool.

11. Related work

There are not many related works that aims to achieve same results. Most of the commercial RTIs provide additional tools and software products that aims to allow easy integration of HLA architecture to existing application. The commercial RTIs are mainly used in army or aircraft simulations therefore the tools are usually aiming to fulfill various specific tasks for army or aircraft simulation. For example:

- Real terrain integration into simulations
- Support for RPR FOM used in army simulations
- Connectivity with various hardware tools that are used for human trainings (e.g. model of a sniper rifle).
- Simulations of audio transferred over air
- Various RTI to other protocol gateways

For more detailed description please visit VT MÄK[16] homepage. The commercial tools are not competitive for this thesis as they are not aimed on providing general purpose solution. Commercial products can provide better connectivity than this thesis, but they are usually targeted for fast integration to some specific environment.

The LVC Game project solves identical problem as this thesis. Their approach is different and not targeted to HLA only. The LVC Game is designed to support multiple lower level protocols, which realize the communication. The main difference is that LVC Game supports specific sets of FOM documents and does not provide complete abstraction

This thesis is direct competitor for GameBots[17] project that is designed to enable external access to Unreal Tournament 2004. Gamebots are using text messages and TCP/IP communication with the engine. The HLA Proxy middleware should be able to outperform GameBots and also allows usage of more advanced features thanks to adopting of the HLA. The biggest difference is that the HLA Proxy provides generic compared to single platform support in GameBots.

It is not possible to identify additional projects related to the same topic. The HLA standard is adopted mainly by large customers for its private projects. Therefore it is not possible to find much information in publicly accessible places.

12. Discussion

The simple idea to enable HLA in 3rd party application that was not designed for HLA support turned out to be a very hard and complex work. The analysis and overcoming of technical difficulties have taken most of the time of work on this thesis.

The game engines that are available today are provided without development documentation. A game engine developer has to reverse engineer the engine code and learn how it is designed by himself. The source code is usually unreadable and lacking comments. This complicates work and implementation of simple features can take very long time.

The cross platform implementation of the middleware was also time consuming because Windows and Linux based operating systems use different set of compilers and source code must be properly written to avoid compiler specific issues (especially when working with C++ templates).

This thesis successfully solved all platform specific issues related to linking and the HLA Proxy middleware is compatible with Windows and Linux based operating systems on both 32bit and 64bit platforms. On windows operating system it is possible to use the HLA Proxy middleware in application that uses both statically and dynamically linked runtime libraries.

Usability of the middleware is quite good. The modular architecture has proven to be a good decision. The HLA Proxy middleware's interface is simple and easy to learn for new developers. Proofs of good interface design are two successful integrations of the HLA Proxy middleware. The prototype implementation has been integrated into two game engines. Both engines are very different. The used version of Source Engine has been released in year 2007. The CryEngine 3 has been publicly released in second half of year 2011. This proves that it is possible to use the middleware under very different conditions.

The biggest achievement of the HLA Proxy middleware is that it has been possible to enable 3rd party application that is not aware of any HLA functionality with transparent HLA support. The application is not aware that its data are being synchronized over HLA with other federates thanks to completely transparent design of the HLA Proxy middleware data classes. Custom database implementation in the middleware can enable storage of data even without access to application's internal data storage.

The middleware is able to adopt new FOM document easily by simple recompilation thanks to automatic source code generation support. Unfortunately support for various data types is currently limited and requires future work to enhance the set of supported data types.

The performance of custom database implementation has overcome expectations and the prototype is able to process decent amount of operations per second. Performance measurements showed that it should be capable of handling reasonable amount of entities to allow a prototyping tool to perform its work.

MAK RTI seems sometimes unstable and the middleware cannot handle such situations well. The HLA federate in the middleware has been implemented according IEEE 1516.1-2010[8] standard and therefore cannot handle unexpected non standardized behavior of the RTI.

This thesis has been successful in addressing all of its main goals.

12.1 Future work

The implementation is still in its early stages of development. It can be used out of the box, but there are many areas of possible improvement. This prototype cannot offer full IEEE1516.1-2010[8] support at this time. More work has to be done, especially in the automatic generation of source code. Current implementation does not support all possible value types that might be very important for future developers.

Next possible continuation is to continue with development of HLA interface. It should be enhanced to provide more HLA features to application that uses the HLA Proxy middleware. More requirements on interface enhancement will come from real deployment in some game engine or prototyping tool.

The HLA Proxy middleware has been designed for federations composed from at most two federates due to limitation of evaluation version of used RTI. It should be possible to switch to an open source RTI implementation that provides C++ interfaces in future. The next step is to start using HLA Proxy middleware in complex federation and improve the middleware to handle it easily.

The very interesting thing might be using the middleware inside a f new project instead of trying to integrate it into complex environment that is not fully extensible. Such deployment can introduce more new ideas for the middleware's use cases and improvements.

13. Bibliography

- 1) Gemrot, J., Brom, C., Plch, T.: A periphery of Pogamut: from bots to agents and back again. In: Agents for Games and Simulations II, LNCS 6525, Springer, pp. 19--37 (2011). The short version of this paper also appeared in Proc. Agents for Games and Simulations, AAMAS workshop pp. 1--19 (2010)
- 2) Kadlec, R., Gemrot, J., Burkert, O., Bída, M., Havlíček, Brom, C., : POGAMUT 2 - A plat-form for fast development of virtual agents' behaviour, In: Proceedings of CGAMES 07, La Rochelle, France, p. 49--53, 2007
- 3) Epic Games: Unreal Tournament 2004 (2004),
<http://www.unreal.com> (8.12.2011)
- 4) Introversion Software: DEFCON, 2006,
<http://www.introversion.co.uk/defcon/> 1.12.2011)
- 5) Valve: Source Engine,
<http://source.valvesoftware.com> (8.12.2011)
- 6) CryTek: CryENGINE 3,
<http://www.crytek.com/cryengine> (8.12.2011)
- 7) IEEE 1516 Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules, 2010
- 8) IEEE 1516.1 Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Federate Interface Specification, 2010
- 9) IEEE 1516.2 Standard for Modeling and Simulation (M&S) High Level Architecture – Object Model Template (OMT), 2010
- 10) Boost libraries,
<http://www.boost.org> (8.12.2011)
- 11) Intel: Thread Building Blocks for Open Source,
<http://threadingbuildingblocks.org/> (8.12.2011)
- 12) Libxml2,
<http://xmlsoft.org/> (8.12.2011)
- 13) Hector Garcia-Molina, Author Central, Jennifer Widom – Database Systems: The Complete Book
- 14) Bohemia Interactive: Virtual Battle Space 2,
<http://armory.bisimulations.com/> (8.12.2011)
- 15) Calytrix Technologies: LVC Game,
<http://www.calytrix.com/products/lvcgame/introduction/> (8.12.2011)
- 16) VT MÄK,
<http://www.mak.com>
- 17) GameBots 2004,
http://pogamut.cuni.cz/pogamut_files/latest/doc/gamebots/ (8.12.2011)

Appendix A. List of examples

| | |
|---|-----|
| Example 4-1 Data manipulation use case | 27 |
| Example 4-2 Event handling | 28 |
| Example 4-3 Time consuming and exclusive operations | 28 |
| Example 5-1 Interaction handler example | 37 |
| Example 5-2 Interaction sending example | 37 |
| Example 5-3 Configuration file..... | 61 |
| Example 5-4 Log output configuration | 63 |
| Example 8-1 The example of Cplayer class extension. | 74 |
| Example 14-1 Unsafe parameter passing by value. | 101 |
| Example 14-2 Unsafe parameter passing by reference. | 101 |
| Example 14-3 Unsafe return by value..... | 102 |

Appendix B. List of tables

| | |
|---|----|
| Table 5-1 HLA component configuration | 35 |
| Table 5-2 Record's metadata structure | 42 |
| Table 5-3 Attribute's metadata | 43 |
| Table 5-4 Member description of BaseRequest class | 43 |
| Table 5-5 Member description of asynchronous request | 44 |
| Table 5-6 Member description of SyncClientRequest class | 45 |
| Table 9-1 Interactions supported by the HL2 federate | 76 |
| Table 9-2 HLA-client supported commands | 77 |
| Table 10-1 Hardware configuration | 80 |

Appendix C. List of figures

| | |
|---|----|
| Figure 1-1 HLA interconnecting various environments over RTI..... | 11 |
| Figure 2-1 Basic concept of HLA | 13 |
| Figure 2-2 Object class hierarchy..... | 14 |
| Figure 2-3 Backwards compatible object model..... | 15 |
| Figure 3-1 Communication between engine and prototyping tool..... | 19 |
| Figure 4-1 Architecture of HLA Proxy middleware | 23 |
| Figure 5-1 Architecture of HLA Proxy middleware | 31 |
| Figure 5-2 Modular architecture | 32 |
| Figure 5-3 Parts of the database component | 38 |
| Figure 5-4 Request dispatching..... | 40 |
| Figure 5-5 Record architecture..... | 46 |
| Figure 5-6 Mapping between keys and records | 47 |
| Figure 5-7 FOM abstraction layer implementation..... | 53 |
| Figure 5-8 Relation between data classes and database records | 54 |
| Figure 5-9 Architecture of output logger | 62 |
| Figure 7-1 Logging via loopback interface | 71 |
| Figure 9-1 Real case deployment scenario..... | 75 |

Appendix D. Introduction to C++, ABI and Runtime libraries

This chapter provides brief introduction to all low level aspects that are related to the shared libraries that are core part of the middleware's design. This chapter can be safely skipped if the reader has enough knowledge about ABI, linkers, C++ temporary object instances. This chapter is important for every developer that plans to extend or modify the middleware itself when the middleware is supposed to be used under non standard conditions. That means:

- Using the middleware with mixed set of runtime libraries.
- Using the middleware with mixed set of compilers

D.1. Linking in the middleware

The component based design of the middleware might look like Microsoft's Component Object Model. In fact it is based on the COM concept. The reason of having this architecture is based on how today's compilers and linkers work. In C++ language there are three options of how to export the specific interface from a shared library:

- By using extern "C" statement.
- By directly exporting the class.
- By combination of abstract interface and factory design pattern.

The first method is the old way in which functions are exported from shared libraries in C language. The main advantage is that the function name is not mangled by the compiler and linking works between executable and library that were built with different compiler versions. There are two limitation of this technique. The first is that we cannot export class methods and we have to hide whole class behind C API. The second is that the library cannot throw exceptions, because the C language does not know anything about exceptions.

Direct export of the class is much better, because it is not needed to solve how the methods are being exported. The class is simply marked for export and the compiler and linker does the rest of the job. This approach allows exceptions to be thrown from inside the class methods, because this is a C++ language construct. The biggest limitation is that this way is not compatible with different compilers or even different version of the same compiler. The reason is different name mangling algorithms used by different compilers. As a result the runtime linker would not find matching imports for binary built with different compiler.

The last technique provides same functionality as second solution but solves the issue in a different way. The first step is to define an interface class, which is abstract with pure virtual methods. The class might contain some generic members and methods without any complications. The second step is to provide implementation of the pure virtual methods in shared library by deriving from the interface class. Both the executable and the library include the interface class' header file. This implementation is working correctly when used with different compilers, because name mangling is not involved during link time. The reason is that all pure virtual functions in interface class are marked as virtual. Therefore the compiler creates virtual

table entries for those methods. The class that implements the interface provides correct hooks to the virtual table.

Every call made through to the interfaces class's method is dispatched via virtual method table and does not involve linking operations. And therefore works correctly with different compilers. The name mangling is now removed, but both compilers have to use same implementation of virtual tables. All compilers are compatible with this solution thanks to the Microsoft Company and its COM model. This functionality is the main requirement of the COM model to be working properly. As the COM model became standard, all compilers that wanted to be compatible with Microsoft Windows operating systems had to ensure they are compatible with this technique.

D.2. Dynamically or statically linked runtime library

On both operating systems (Windows or Linux based) it is possible to link the binary statically with the runtime libraries. This is one of the biggest complications in middleware's design. The problem is same as for regular shared or static libraries. Unfortunately the runtime libraries provide critical functionality for the application. If the runtime library does not work correctly then very strange things happens. It might be even impossible to debug such issues if they are caused by calling conventions (see next chapter), because the issue happens at very low level code generated by the compiler.

The statically linked runtime generates bigger binary output. However the resulting binary does not depend on dynamic runtime libraries. This allows easier deployment of application. For example, the application can be transferred between various Windows versions without problems. The limiting factor is that memory management must be secured. It is not possible to allocate dynamical data from one runtime instance and then releasing them in other instance. In such case the application will surely crash. It is very easy to encounter such problems without precise design of APIs shared between binary and its libraries.

The dynamically runtime provides much simpler memory management. As all participants (exe, dlls) share the same runtime library, it is safe to allocate/deallocate dynamical data at any place in the source code. This is true as long as all of the participants link to same library. The other benefit of having dynamically linked runtime is correctly working Run-Time Type Information. As long as all participants share the same runtime library the RTTI is same for all of them. If you use statically linked runtime some things like `dynamic_cast` might break or not working as expected. This problems are usually called the DLL Hell. The debugging of such errors is very complicated and almost impossible without knowledge of how these things are implemented. The debugger is very often confused by such reports and stops in weird locations in the source code, or does not provide any debugging at all.

The disadvantage of dynamically linked runtime and other libraries lies in problems with deployment and updates to the environment. The system wide library can be changed at any time and if the update changes the library behavior the middleware and application will simply break.

The main idea of the middleware was to provide modular architecture that can be easily extensible. This means that new plug-in's shared libraries may be added at any time. Also there is no need to rebuild the whole middleware every time some plug-in changes. Therefore whole middleware is using dynamically linked runtime. The second reason is that middleware depends on third party libraries that use

the dynamically linked runtime too. Therefore it is expected that application will use the dynamically linked runtime.

One of the goals is to integrate the middleware to the Source Engine. The Source Engine uses statically linked runtime libraries. The result is that the HL2.exe process is going to be extended by multiple other libraries, which use dynamically linked runtime. The process is going to be running in so called mixed mode. The Source Engine provides its own memory management and overloads some of the standard runtime methods like malloc/free. It is not possible to easily identify what is different in the Source Engine and therefore the middleware needs to have well designed API to behave as much safe as possible when dealing with dynamically allocated memory.

If the middleware is going to be integrated to newly created project without its limitations then always use dynamically linked runtime whenever it is possible to do so. This is the only safe way to prevent some surprising runtime failures.

D.3. Calling conventions

To understand calling conventions it is necessary to look at the source code from the point of view of a machine. A machine does not see any higher level types. It can operate only with stack, memory and CPU registry. It does not support calling methods, just calling of functions. The Application Binary Interface (ABI) defines rules to which every binary output from compiler must adhere to be working on specific hardware. Calling conventions are part of this binary interface. They define the protocol that is used during the function calls. The calling convention defines responsibilities of caller and callee, way how arguments are passed to the function and way how the return values are passed back. It is not possible to call a function inside shared library with different calling convention than the one it has been built with.

Understanding of various calling conventions in C++ is very crucial. Every today's compiler has some set of supported calling conventions. The problem is that some keywords are interpreted in different ways by different compilers. The result is that it is not safe to mix multiple compilers to build the middleware, but it does not mean that it is not possible.

In Microsoft Visual Studio it is possible to define default calling convention for whole project, but this applies only to functions. Usually the default configuration is to use `__cdecl` convention. The `__cdecl` calling convention is C style function call. The arguments of the function are pushed on top of the stack in reverse order (the last argument is at the top). This allows calling of functions with variable count of arguments. The `__cdecl` also defines that after the call the caller is responsible for stack cleanup.

The class methods are plain functions, which takes one extra argument that points to the instance of the class. The default configuration is to use `__thiscall` convention. This is important for the middleware, because data.dll is exporting its classes directly. The `__thiscall` convention in Microsoft's compilers does not push pointer to the class on the stack as one of the function arguments. The ECX CPU register is used to pass the pointer. This is a problem as gcc compilers do not support this type of function calls, they push the pointer to the stack. The `__thiscall` calling convention support has been added to the gcc version 4.6. Currently older gcc compilers cannot link against the data shared library, unless it has been compiled with gcc.

The only known solution is to change the calling convention of exported class methods to `__cdecl`. By using the `__cdecl` convention the Microsoft's compilers are

forced to generate gcc compatible calls. The middleware currently uses `__thiscall` as the whole project and Source Engine is being compiled with the same compiler. If there is need to use the middleware with mixed set of compilers the calling conventions might need reconfiguration.

For more information about various other calling conventions please visit the Microsoft's MSDN library.

D.4. How to make API calls safe in mixed runtime environment

The middleware's API is a result of analysis of calling conventions, memory management and linking. The API should be safe enough to work under all conditions. The implementation is at some places more complicated than it is required to. In most cases this is not a bug, but intentional implementation and every modification to such parts of code should be done very carefully.

The first principle of the API design is "keeping all memory management at single place". That means user can't allocate instances and free them by using new or delete operators. If he does then things most probably break.

The factory design pattern is used at many places to instantiate user accessible classes. This way, allows having memory allocation being done from the factory instance and therefore in same instance of the runtime library. Usually the implementation of the interfaces is completely hidden before users and such classes are not exported from plug-in's shared libraries. The non accessible implementations guarantee that user can't accidentally instantiate an interface implementation under some unknown conditions.

Factories are responsible for correct allocation of memory resources. All interface implementations resides in the same plugin as their factory. Therefore the `release()` method, which is part of every interface, is going to map to correct runtime. This design is similar as the COM. It has been verified by years of usage, so there is no reason to try design different solution.

Every interface has some methods that allows to pass data from/to the interface implementation. Every such function must be designed with memory safety in mind. Therefore the default responsibilities (if not mentioned otherwise in manuals) are that the Caller is responsible for allocation and releasing of all data passed to the middleware interfaces. If the interface implementation needs to store the data for longer time it must create its own copy. If the middleware's user receives some data from the interface instance, they must be treated as volatile. The only thing the user can do is to create his/hers own local copy.

The middleware APIs should only pass simple types or pointers around. Working with complex types requires extreme caution. Always consider whether the complex type is really necessary and whether it can possibly cross the runtime boundary. Good example is the data library that provides interface that is safe to use across mixed runtimes boundaries.

To completely understand the issue the rest of the chapter explains various problems based on compiler behavior. All following cases are covered in detail:

- Passing complex type by value.
- Passing complex type by reference.
- Returning complex type by value.
- Returning complex type by reference.

Passing complex type by value is extremely unsafe operation in mixed runtime environment.

```
void fnc (std::string str)
{
    ... do something with str
}
```

Example 13-1 Unsafe parameter passing by value.

The Example 13-1 shows hidden problem cause by the code that will be generated by the compiler. The temporary copy of string object will be created before the call and passed to the function as an argument. The C++ language destructs the temporary variables when they are not needed. It means when the expression, which the variable is located in, has been evaluated or when the control flow reaches the end of control block. The secret problem here is caused by the ABI of C++. Complex classes are never passed on stack, they are always allocated dynamically and only the pointer to them is passed on the stack. In this case the Caller of fnc allocates temporary instance, but the Callee is going to destroy it at the end of function. As the memory pointer is being freed from different runtime the application is going to crash. As a result, it is not possible to pass complex types by value on mixed runtime boundaries. This behavior is not going to change, even when using `__cdecl` calling convention.

```
void fnc (std::string &str)
{
    ... do something with std
}
```

Example 13-2 Unsafe parameter passing by reference.

The Example 13-2 shows different approach. Let's pass the instance by reference. At this time the problem with passing by value is gone. The Callee cannot call destructor as he is only holding the reference. The string class is going to be destructed when the Caller reaches end of his expression or control block that triggered the call. But there is another problem. The argument has been passed as a writable reference to the class. This allows the fnc to modify the argument. The string is keeping internal pointer to its data. If the fnc assigns new value to the string the str is going to release and allocate new buffer. This is the first point when the application is going to crash, because the buffer has been allocated from within Callers runtime. Such issue is easy to debug, the application will crash at the line of code that contains assignment operation.

The problem can become even much worse. The string can be instantiated as empty and therefore it won't allocate its buffer. Then the string is modified, but this time the free is not called as the string has no buffer yet. All assignment operations are going to be successful inside fnc, because all memory allocations/frees are being done by same runtime. Once the function returns the string is working correctly. But when someone tries to assign or destroy the string the application will crash, because the internal buffer is allocated in different runtime. It can happen probably much later after the fnc call and it won't be seen as a failure in fnc at all during debugging.

The result is, not all calls done by reference are safe. Always be sure that some method call won't allocate or free memory inside the class. To prevent such

mistakes always pass references as const. But it can't work in every case (mutable attributes of class are writeable from const calls).

```
std::string fnc ()
{
    ... do something
    return str;
}

std::string str_one;
str_one = fnc();
std::string str_two = fnc();
```

Example 13-3 Unsafe return by value.

The Example 13-3 shows the function that returns complex class by value. This example is much trickier than the previous ones. Basically there are two scenarios and both of them will have issues with mixed runtime.

When the user calls the method, the application is going to crash immediately after the call to the fnc. The first language construction enforces usage of operator=. Therefore the compiler constructs anonymous temporary variable holding the result of the call, then calls assignment operator and then destroys the temporary instance. At this point the application tries to destroy buffer in temporary variable that has been allocated in different runtime library.

However the second way seems to be working correctly. As was explained later it is only a matter of time until the next assignment operation or destruction occurs. The difference is that this time the assignment worked. The reason is in optimization done by the compiler and usage of copy constructor. Many of today's compilers have pass-by-value optimization. The result is that the compiler extends the function arguments by one and uses this hidden argument to pass pointer for pre-allocated memory for the result. The return statement then simply calls the copy constructor and initializes the memory structure. At this point incorrect internal buffer is passed to the string class. Unfortunately there is no temporary variable so no destruction occurs after the application will crash later.

The last option of transferring values from function is to return only a reference to the class. This is the most safe and preferred method in the middleware's API. However it has significant limitation. It is not possible to return values of internal function variables. The instance must exist after the exit from fnc. By returning a reference the construction of resulting variable is done completely on the caller side.

All of these examples are intentionally using std::string class. That is because the class is exported from the runtime libraries. As a result the caller and callee call different methods. One of possible solutions to complex types problem is to have the type in external library so both sides the caller and the callee will call constructors from the same library. In this way everything will work correctly. The main complications are caused by STL classes that are being passed across the mixed runtime libraries.

These limitations are required for the integration to the Source Engine, which uses statically linked runtime libraries. When building application linked dynamically to the runtime libraries, the restrictions no longer apply. The goal of this thesis is to provide general solution. Therefore the main design is considering that whole process shares the same dynamically linked runtime libraries. However one of the goals is to connect the Source Engine, so there are some unnecessary safety mechan-

isms in place to make it possible. It is safe to remove them when the limiting conditions do not longer apply.

D.5. Binary compatibility

There is one thing that cannot be guaranteed and that is binary compatibility. Every method in the component's interface, which takes pointer to data structure as an argument, can get easily broken. For example the structure passed through the interface gets resized. It is still possible to pass pointer to the structure through the interface, but it can result in undefined behavior. Every interface developer should define not only component's interface but also data structures that will be passed through the interface's methods. Every modification of such a structure should be done after impact analysis of the change. In most cases it is better to rebuild whole projects to avoid binary compatibility issues. Few classes that are passed through the interface are designed using C++ templates. This also adds new requirement and that all of the parts and application that uses the middleware must be rebuild by the same compiler. This is required to ensure that all template based classes expand in the same way and do not break binary compatibility.

It is possible that plug-ins or plug-in management will evolve in time. It is necessary to control binary compatibility between plug-ins and plug-in management layers. This is done by versioning of the plug-ins. Every new version of plug-in management shall increment the plug-in engine version number. This number is hidden behind constant defined in the SDK. The constant is then used during build of both plug-in manager and plug-ins. When plug-in management tries to load plug-in it checks whether the plug-in version is the same as the plug-in management version. This mechanism minimizes the possibility that incorrect plug-in is used during runtime. The value is compared by using equality comparator. That means the plug-in must match the version of used plug-in engine. At this point there is no backward compatibility guaranteed between higher versions and older versions. The check can be changed in the future when needed, however it will have impact on future development of the middleware

The versioning can be used to synchronize data structures between all release components. Every significant change that impacts other layers should increment the version number. In the end it is still developer responsibility to provide correct implementation/builds of all of the components. This mechanism exists only as a basic protection, but cannot guarantee that all of the pieces are correctly built.

The method of how to deliver builds of the middleware to the customer is not part of this thesis. Every project can choose its own best suitable release model and the plug-in versioning may/may not play a part in this role. It is possible to deliver more sophisticated component versioning solution, but it is out of scope of this thesis.

Appendix E. Setup MAK RTI

The MAK RTI setup should be straight forward, but it is not. This chapter describes all of the issues that were discovered during implementation and also provides step by step instructions how to setup your HLA environment correctly.

The minimal MAK RTI version is 4.0.4, which is the first official version that supports HLA Evolved (IEEE 1516-2010) standard.

E.1. Microsoft Windows

The installation of RTI middleware for Windows platform is very simple, because MAK Company provides windows installer. Just follow the steps in installation wizard. Please read following notes, before proceeding with the installation of the RTI.

Do not install Windows Live pack, because it can interfere with your MAK RTI installation and effectively prevents the RTI startup. This root cause is unknown, but it looks like the Live pack installation delivers some new shared libraries that cause very long time startup of the RTI. As there is no root cause of the problem, it is possible that the problem no longer appears.

The MAK RTI startup can take 10s of minutes, due to the bug in hostname lookup on Windows 7. Always install at least version 4.0.4 (provided with the thesis on DVD), which contains fix for this issue.

E.2. Fedora 16

The only one Linux based operating system supported from the MAK Company is Red Hat Enterprise Linux 4 and 5. Fedora compatible version of the RTI is 3.4, which is currently unsupported by the HLAProxy middleware (due to the lack of HLA Evolved support). The version 4.0.4 for RHEL 5 has been tested successfully on different Linux distributions (Fedora 16, Ubuntu 10.10), however 100% compatibility is not guaranteed by the RTI vendor.

The biggest problem is that the HLA Proxy middleware has issues when being built by g++ 4.1.2 (in CentOS 5), most probably due to the weak implementation of the RTTI. Even though it is possible to compile the middleware without problems, it does not run correctly. The decision has been made that the thesis will be run on Fedora 16 with more recent version of g++ 4.6.2. Some parts of the middleware are not working there (RTIsy due to SSL problems). However the RTI works correctly and without problems.

The installation is straight forward; just extract the linux archive to any location you want. Then setup environment variable RTI_HOME to point to the root of the RTI directory.

The GUI of the RTI depends on the Qt4 libraries, which are usually installed on latest version of Linux distributions.

Appendix F. Building of the middleware

This appendix provides step-by-step instructions of how to setup build environment and how to build/install the middleware.

F.1. Windows build environment

The official supported version of Windows operating system is Windows XP and newer. The limiting factor on Windows operating systems is the version of Microsoft Visual Studio that is used to build the middleware. The only supported version is the Microsoft Visual Studio 2008 Service Pack 1.

F.1.1. Building boost libraries

The boost for windows is distributed on the DVD as a zip archive or can be downloaded from www.boost.org. Unzip the archive to some convenient location and then build the shared libraries by using:

```
.\bootstrap.bat  
.\b2 variant=debug,release link=static,shared threading=multi runtime-link=shared
```

If you have Microsoft Visual Studio 2010 installed it will be used by default. The toolset=msvc-90 option can be used to build the boost libraries using Microsoft Visual Studio 2008. However it is possible that output binaries are not going to be named properly and some renaming has to be done by hand.

F.1.2. Altova XSLT processor

The Altova XML tools are distributed for free and support XSLT version 2.0. The installation of Altova tools is simple by using installer provided on the DVD. The easiest option is to install the Altova tools to default folder (C:\Program Files).

It is possible but not so easy to modify tools location in the middleware. To change the location of the Altova tools requires two following steps:

1. Modify custom pre-build step of the data project
2. Modify the data\xslt\data_vcproj.xml and all paths inside

The modification of data_vcproj.xml file is essential. Otherwise the build will be broken.

F.2. Building the middleware on Windows

The middleware is distributed as a solution for Microsoft Visual Studio 2008 (VC9). There are no extra requirements for building the middleware, except configuration of correct include and library paths.

It may be required to rebuild the data project two times. Because the first run will automatically generate source files and second attempt will build them. This is a side effect of very simple project file generation mechanism implemented for the data shared library.

F.3. Linux build environment

The official supported Linux platform for MAK RTI is the Red Hat Enterprise Workstation 5. It is possible to build the HLA Proxy middleware on other Linux distributions, but it is unsupported build environment.

The DVD contains script that will setup CBE for Fedora 16. The only required steps are extracting the MAK RTI and configuration of the include and library paths

F.3.1. Saxon XSLT processor

The middleware requires XSLT processor that is compatible with XSLT 2.0 standard. The build environment supports Saxon parser. The Saxon processor is written in Java language and therefore should be compatible with any Linux distribution that is capable of running Java Run Time Environment from Oracle.

The first step is to install JRE 1.6 from Oracle. For convenient installation the JRE is distributed on the DVD attached to this thesis. Use following commands to install the JRE:

1. `chmod +x jre_____todo` concrete file name
2. `./jre_todo` concrete file name

The next step is to extract saxon9he.zip from the DVD to some local directory. For convenient usage of saxon it is wise to create a more convenient wrapper. Create an executable script in your `/usr/bin` directory with following contents:

```
Java -jar /<saxon installation>/saxon9he.jar $*
```

This configuration allows to easily setup saxon to work with the thesis without knowing where is the java and saxon.jar file.

F.4. Building the middleware on Linux

The external dependencies on Linux are solved much more easily due to the package management support. Please install following packages before you start to build:

- libxml2-devel
- tbb-devel
- gcc-c++
- saxon
- java

The first library is available in repositories of almost any distribution. The Intel's Thread Building Blocks might not be present. The TBB is distributed on the attached DVD for cases where it is not possible to install it by using package management.

Prior to building of the middleware install all of prerequisites. Please adjust the make files include and library paths to point to the correct location of the external dependencies.

The build itself consist from following steps:

1. Automatic generation of source code
2. Pre installation of public header files to shared area
3. Build of all components

Appendix G. The HLAProxyMod

G.1. Building of the mod

The HL2 customized mod can be built after successful build of the HLProxy middleware. The Visual Studio project is preconfigured to look for includes inside the workspace. However few dependencies need to be configured:

- Boost
- MAK RTI

The Visual Studio's solution is preconfigured to copy resulting client.dll and server.dll to the Steam's mod directory. The path needs to be corrected if the HLProxyMod is located in different directory. Otherwise the outputted libraries have to be copied manually.

The following files have to be copied from the middleware's build directory to the mod's bin directory:

- plugengine.dll
- config.dll
- source.dll
- hla.dll
- db.dll
- data.dll
- config-hl2.xml
- aifom.xml

Following libraries has to be present in Source Engine SDK folder:

- tbb_debug.dll
- boost_date_time.dll
- boost_system.dll
- boost_thread.dll
- data.dll

The config-hl2.xml file prototype is located in HLAProxy/workspace/input-files. There should be no other modifications required. The user is free to customize the configuration to suit his/hers specific needs.

G.2. Running the mod

The mod can be started only when the user is signed with valid Steam account to the Steam network. Without valid account the Steam allows to run only single-player mode games. The HLAProxyMod is modification of multiplayer-game so it is not possible to run it without having Steam in online mode.

The MAK RTI shows pop-up window, when there is no default connection configured or the connection is not valid. Therefore it is necessary to run the HL2 in windows mode, otherwise the application will appear as deadlocked. To enable windowed mode follow next steps:

1. Open Steam application and navigate to the Library.
2. Select HLAProxyMod and open its properties via context menu (by using right mouse button).
3. Choose “Set launch options”
4. Insert “-allowdebug -novid”

The HLAProxyMod can be launched via Steam or directly from Microsoft Visual Studio. Before starting the HLAProxyMod from Microsoft Visual Studio please ensure that your debugging is setup correctly. The correct way how to launch the mod is by using following project options:

| | |
|-----------------|---------------------------|
| Command: | <Source SDK 2007>\hl2.exe |
|-----------------|---------------------------|

| | |
|-------------------|---|
| Arguments: | -allowdebug -novid -game <mod path>\hlaproxymod |
|-------------------|---|

The HL2 instance serves as a HLA federation master, it is important that it must be started as the last of all federates, which participates in the federation execution.

